I hereby certify that this correspondence is being deposited with the U.S. Postal Service with sufficient postage as First Class Mail in an envelope addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, VA , 22313-1450, on:

Date: 9/7/05 By: [signature]

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Appl. No.	:	**09/784,699**	**Confirmation No.**	:	**3149**
Applicant	:	**Benitez et al.**	**TC/A.U.**	:	**2157**
Filed	:	**February 14, 2001**			
Examiner	:	**Hussein A. El Chanti**			
Docket No.	:	**30126-8009.US01**	**Customer No.**	:	**22918**

Declaration of Prior Invention Under 37 C.F.R. § 1.131

Commissioner for Patents
P.O. Box 1450
Alexandria, VA

I. This Declaration establishes invention prior to September 26, 2000.

II. This Declaration is being made by Daniel T. Arai, i.e., a named inventor of the above-identified patent application.

III. Conception: Prior to September 26, 2000, we conceived the inventions currently presented in independent claims 1, 16, and 31, of the above-identified patent application. A list of these claims is attached hereto as Exhibit A. Claim 1 is exemplary of an embodiment of the inventions. Exhibit B includes a listing of files related to a product that is representative of the embodiment claimed in the exemplary independent claim 1. Exhibit B includes versions of software and documentation that were created in a Content Management System (CMS) prior to September 26, 2000. The dates of each file have been redacted.

Exhibit C includes a subset of content from the CMS as of September 25, 2000. The content is from files listed in Exhibit B, which are entitled Exhibit C1 through C18. Exhibit C correlates to the exemplary independent claim 1. These

correlations are for the purpose of example only, and not intended to limit the scope of the claims. TABLE 1 provides a rough correlation between Exhibit C and, for example, independent claim 1:

TABLE 1

<table>
<tr><th>EXHIBIT C (Examples only)</th><th>CLAIM 1</th></tr>
<tr><td>
C2)

Functionality (pg 1)
The Client Network Interface (CNI) provides the interface for sending message to servers and provides threads for receiving responses and dispatching them appropriately. (para. 1).

Asynchronous Server Calls (pgs. 5 – 6)

The network send thread is periodically awoken, and it coalesces requests off the NW request queue and sends them to the server (pg. 6, para. 1).
The network receives thread waits for responses to come back from any server. (pg. 6, para. 2).
Finally, the response dispatch thread pulls responses off the response queue, and handles the work of dispatching them appropriately. (pg. 6, para. 3)

C6)

Estream client network interface

Handles requests from Estream cache manager
Handles protocol interface to/from server

C7) Diagram illustrating structure of server streaming of application programs across a computer network while executing application programs on an Estream client.

C17) Abstract and descriptions of CORBA illustrate an implementation of a server framework for Estream. (pg. 1).

C18) Diagram illustrating structure of server streaming of applications programs across a computer network while executing application programs on an Estream client.
</td><td>
(a) A process for intelligent server streaming of conventionally coded streamed application programs across a computer network while concurrently executing said streamed application programs on a client in a computer environment, comprising the steps of:
</td></tr>
</table>

(pg. 7)	

C1)
• AIMInstallApplication Prototype (pg. 16)
  o Step 5. Initializing the profile and prefetch data for this app.
 	
C5)
• Installation of application
  o Contact designated App Server using id info, download meta-data about app, potentially including registry/DLL/filesys spoofing info, prefetching info, initial cache contents for app.
  o Perform initial installation & setup for app, after checking system for previously installed version of app & issuing any appropriate warnings.
 	
C10)
• Functionality (pg. 1)
  o The AppInstallBlock is a block of code and data associated with a particular application. This AppInstallBlock contains the information needed by the Estream client to "initialize" the client machine before the Estream application is used for the first time.
 	
C11) Data format of the AppInstallBlock.
	(b) downloading an initial portion of a streamed application program on said client wherein said streamed application program comprises page segments and wherein said initial portion of said streamed application remains on said client after terminating execution of said streamed application by said client;

C7) Estream client-server diagram comprising an application server. C8) • Finding an application server for a volume. (pg. 2). o The SLM will tell the client which application servers currently provide each volume. It may be necessary for the client to periodically poll the SLM to get up-to-date information about the state of the application servers.	(c) providing an application server;

C9)
• **Technical description of the invention. (pg. 12)**
  o **An application file server: Responds to requests by client application cache manager for portions of application's files and directory structure on the server. Transmits compressed information for better bandwidth utilization. (pg. 12, item #4).**
 	
C16)
• **Functionality (pg. 1)**
  o **The primary job of the App Server is to service client requests for application data blocks. (pg. 1, para. 4).**
  o **The App Server serves data derived from Estream Sets. (pg. 1, para. 5).**
 	
C18)
• **Application Server (pg. 26)**
  o **The application server is there to handle read requests for files accessed by Estream clients. Any file accessed on a client through the EFS can have this read request passed to an app server.**	

C7) Estream client-server diagram illustrating transmission of Estream sets.
 	
C9)
• **The process of building a new set of request replies for an application is called building an application stream set. (pg. 15, 3rd full paragraph).**
• **An application stream set contains (pg. 15, 3rd full paragraph):**
  o **A unique name of the application for reference purposes,**
  o **An index table used to quickly determine which reply to return for a given request,**
  o **The set of all possible request**
	(d) partitioning said streamed application program into said page segments on said application server;

replies. • **The application stream set is built in the following manner.... (pg. 15, 4th full paragraph).** **C12)** • **The Estream Builder is a software program. It is used to convert locally installable applications into a data set suitable for streaming over the network. The streaming-enabled data set is called the Estream Set. This document describes the procedure used to convert locally installable applications into the Estream Set. (pg. 1, para. 1).** **C13) Estream Builder data flow diagram illustrating the conversion of locally installable applications into Estream sets.** **C14)** • **Functionality (pg. 1)** o **The Estream Application Builder Package Manager is responsible for packaging data gathered from the Installation Monitor, the Profile Manager, and the Upgrade Monitor into a set of data called the Estream set. (pg. 1, para. 1).** **C15)** • **Functionality (pg. 1)** o **The Estream set is a data set associated with an application suitable for streaming over the network. The Estream set is generated by the Estream Builder program. This program converts locally installable applications into the Estream set. This document describes the format of the Estream set. (pg. 1, para. 1).** o **Diagram illustrating format of the Estream set. (pg. 5).**	
C7) Estream client-server diagram illustrating transmission of Estream sets to	**(e) wherein said application server streams said page segments to said client upon said**

Estream client. **C9)** • **From the point of view of the client and its operating system, the application is resident locally on the client; the execution controller negotiates with an appropriate server to allow the client to obtain (as needed) segments of the associated application files located on the servers. (pg. 12, para. 2).**	**client's request;**
C9) • **The execution controller is given an argument indicating which application is to be executed. From the point of view of the client and its operating system, the application is resident locally on the client; the execution controller negotiates with an appropriate server to allow the client to obtain (as needed) segments of the associated application files located on the servers. (pg. 12, para. 2).** • **If a server accepts the task of serving the application to the client, the execution controller passes the application access request on to the application remote file interface code. This code allows the client to reference file and directory information associated with the remote application as if it resided on a local physical disk device. (pg. 13, 3rd full paragraph).**	**(f) wherein the user starts said streamed application program as if said streamed application program were fully installed on said client;**

C5)
• **Execution of application (pg. 2)**
  o **Send unique certificate for application to appropriate ASP DRM Server, get back id for closest/best App Server & session id.**
  o **Contact designated App Server using id info, request file system data as necessary. Respond to running application's requests,**
	(g) wherein specific page segments are requested by said client's file system during execution of said streamed application program such that said streamed application program begins execution on said client prior to downloading all of said page segments; and

collect usage data.
 	
C9)
• **The client's operating system begins executing the requested application located remotely on a server. The operating system memory-maps the application and begins executing it, with the application remote file interface code obtaining control whenever the client system's page fault handler determines that the application's page is located on the remote disk drive. The page fault handler asks the application remote file interface code to place the appropriate page data in main memory. The application remote file interface code sends a request to the cache manger for the desired data. (pg. 13, 4th full paragraph).**
 	
C12)
• **Data flow description (pg. 6)**
  ○ **The OS loads the application executable into memory and runs the executable. (pg. 7, step 12).**
  ○ **The executable file image is loaded into memory and starts executing. The application files will continuously be loaded into memory as needed. (pg. 7, step 13).**	

C3) • **Functionality (pg. 1)** ○ **The cache manager manages the on-disk cache of file system data, and the in-memory data structures for managing this cache. (pg. 1, para. 3).** • **Diagram illustrating overall client architecture comprising cache elements. (pg. 8).** • **Implementation of the cache manager. (pgs. 11 – 17).** **C4)**	**e) storing said page segments in a cache on said client.**

• **Cache organization (pg. 1)**
  ○ **The cache will be contained in 2 or more files. One file will contain the cache indices, and one or more files will contain the data blocks for cached files.**
 	
C5)
• **Execution of application (pg. 2)**
  ○ **Contact designated App Server using id info, request file system data as necessary. Respond to running application's requests, collect usage data. Cache portions of applications, file system info, & user preference info.**	

IV. <u>Diligence:</u> We diligently constructively reduced the invention to practice on Nov. 6, 2000. Attached, with dates redacted, as Exhibits D1 through D5 (collectively "Exhibit D") are exemplary documents produced between September 26, 2000 and constructive reduction to practice. These documents are in chronological order, and have redacted dates which occurred at irregular intervals but without interruption extending from our conception of the invention to our constructive reduction to practice of the invention. Exhibit D is as follow:

a) D1: Estream 1.0 planning document

b) D2: Estream server component framework low level design

c) D3: Estream set format low level design

d) D4: Estream 1.0 high level design

e) D5: Estream web server load monitoring applet low level design

Exhibit D correlates to the exemplary independent claim 1. These correlations are for the purpose of example only, and not intended to limit the scope of the claims. TABLE 2 provides a rough correlation between Exhibit D and, for example, independent claim 1:

TABLE 2

EXHIBIT D (Examples only)	CLAIM 1

D1) Server group time estimates for implementation.
 	
D2)
• **Functionality (pg. 1)**
  ○ **The Server Component Framework provides a common basis on which server components are implemented. The framework provides a number of services such as common server initialization and configuration, messaging, state management, logging, and error handling. (pg. 1, para. 1).**
 	
D5)
• **Functionality (pg. 1)**
  ○ **One of the requirements for the Estream web server is a facility for monitoring server load. Per this document, this facility will be provided by a graphical load-monitoring applet that will be available for deployment at customer sites as part of the Estream web server installation.**
	(a) A process for intelligent server streaming of conventionally coded streamed application programs across a computer network while concurrently executing said streamed application programs on a client in a computer environment, comprising the steps of:

D1) • **Content** ○ **AppInstallBlk structure time estimate for implementation.**	**(b) downloading an initial portion of a streamed application program on said client wherein said streamed application program comprises page segments and wherein said initial portion of said streamed application remains on said client after terminating execution of said streamed application by said client;**
D1) • **Server group** ○ **App Server time estimate for implementation.**	**(c) providing an application server;**
D1) • **Content** ○ **File access monitor time estimate for implementation.**	**(d) partitioning said streamed application program into said page segments on said application server;**

<table>
<tr><td>

Packager time estimate for implementation.
Estream distribution time estimate for implementation.

D3)

Functionality (pg. 1)
The Estream set is a data set associated with an application suitable for streaming over the network.
Estream set format diagram (pg. 6).

</td><td></td></tr>
<tr><td>
D4)

App server
The application server is there to handle read requests for files accessed by Estream clients. (pg. 10).

</td><td>(e) wherein said application server streams said page segments to said client upon said client's request;</td></tr>
<tr><td>
D4)

Overview

A small client 'player" program to allow local execution of applications that reside on the servers. (pg. 1)
The user will now see standard shortcuts for subscribed applications, exactly as though the app were installed locally. (pg. 2, 6th box)

</td><td>(f) wherein the user starts said streamed application program as if said streamed application program were fully installed on said client;</td></tr>
<tr><td></td><td>(g) wherein specific page segments are requested by said client's file system during execution of said streamed application program such that said streamed application program begins execution on said client prior to downloading all of said page segments; and</td></tr>
<tr><td>
D1)

Client group
Estream cache manager time estimate for implementation.

D4)

Client components (pg. 7)
ECM: the Estream cache manager. This is the user-space component that handles requests from the Estream File System

</td><td>e) storing said page segments in a cache on said client.</td></tr>
</table>

Driver, and manages the on-disk and in-memory cache of file contents.	

V. We hereby declare that all statements made herein of our own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, (18 U.S.C. §1001) and that such willful false statements may jeopardize the validity of this application or any patent issued thereon.

Daniel T Arai

Daniel T. Arai

Date August 12, 2005

Exhibit A

The Claims

1. A process for intelligent server streaming of conventionally coded streamed application programs across a computer network while concurrently executing said streamed application programs on a client in a computer environment, comprising the steps of:

downloading an initial portion of a streamed application program on said client wherein said streamed application program comprises page segments and wherein said initial portion of said streamed application remains on said client after terminating execution of said streamed application by said client;

providing an application server;

partitioning said streamed application program into said page segments on said application server;

wherein said application server streams said page segments to said client upon said client's request;

wherein the user starts said streamed application program as if said streamed application program were fully installed on said client;

wherein specific page segments are requested by said client's file system during execution of said streamed application program such that said streamed application program begins execution on said client prior to downloading all of said page segments; and

storing said page segments in a cache on said client.

2. The process of claim 1, wherein said streamed application program is not recompiled, rewritten, or rebuilt for this specific delivery mechanism.

3. The process of claim 1, wherein said client manages said cache by purging page segments that are stale or not needed.

4. The, process of claim 1, wherein said client does not request page segments of said streamed application program that already reside in said cache.

5. The process of claim 1, further comprising the step of:

providing a subscription server; and

wherein the user subscribes or unsubscribes to streamed application programs with said subscription saver.

6. The process of claim 1, further comprising the step of:

providing a license server; and

wherein said client obtains an access token for a requested streamed application program from said license server if the user has a valid subscription to said requested streamed application program.

7. The process of claim 6, wherein said access token contains an expiration tag.

8. The process of claim 6, wherein said access token is securely encrypted.

9. The process of claim 6, wherein said client passes said access token to said application server before requesting page segments of said streamed application program.

10. The process of claim 6, wherein if said license server fails said client automatically switches to another license server.

11. The process of claim 1, further comprising the step of:

providing a profile information database characterizing the typical page segment needs of each streamed application program on said application server.

12. The process of claim 11, wherein said profile information database is updated dynamically as page segments are requested from said application server.

13. The process of claim 11, wherein said client prefetches page segments of said streamed application program from said application server based on the profile information of said streamed application program.

14. The process of claim 11, wherein said application server pushes page segments of said streamed application program to said client based on the profile information of said streamed application program.

15. The process of claim 1, wherein said client performs load balancing among a plurality of application servers for page segment requests.

16. An apparatus for intelligent server streaming of conventionally coded streamed application programs across a computer network while concurrently executing said streamed application programs on a client in a computer environment, comprising:

a module for downloading an initial portion of a streamed application program on said client wherein said streamed application program comprises page segments and wherein said initial portion of said streamed application program remains on said client after terminating execution of said streamed application program by said client;

an application server;

partitioning said streamed application program into said page segments on said application server;

wherein said application server streams said page segments to said client upon said client's request;

wherein the user starts said streamed application program as if said streamed application program were fully installed on said client;

wherein specific page segments are requested by said client's file system during execution of said streamed application program such that said streamed application program begins execution on said client prior to downloading all of said page segments; and

a module for storing said page segments in a cache on said client.

17. The apparatus of claim 16, wherein said streamed application program is not recompiled, rewritten, or rebuilt for this specific delivery mechanism.

18. The apparatus of claim 16, wherein said client manages said cache by purging page segments that are stale or not needed.

19. The apparatus of claim 16, wherein said client does not request page segments of said streamed application program that already reside in said cache.

20. The apparatus of claim 16, further comprising:

a subscription server; and

wherein the user subscribes or unsubscribes to streamed application programs with said subscription server.

21. The apparatus of claim 16, further comprising:

a license server; and

wherein said client obtains an access token for a requested streamed application program from said license server if the user has a valid subscription to said requested streamed application program.

22. The apparatus of claim 21, wherein said access token contains expiration tag.

23. The apparatus of claim 21, wherein said access token is securely encrypted.

24. The apparatus of claim 21, wherein said client passes said access token to said application server before requesting page segments of said streamed application program.

25. The apparatus of claim 21, wherein if said license server fails said client automatically switches to another license server.

26. The apparatus of claim 16, further comprising:

a profile information database characterizing the typical page segment needs of each streamed application program on said application server.

27. The apparatus of claim 26, wherein said profile information database is updated dynamically as page segments are requested from said application server.

28. The apparatus of claim 26, wherein said client prefetches page segments of said streamed application program from said application server based on the profile information of said streamed application program.

29. The apparatus of claim 26, wherein said application server pushes page segments of said streamed application program to said client based on the profile information of said streamed application program.

30. The apparatus of claim 16, wherein said client performs load balancing among a plurality of application servers for page segment requests.

31. A program storage medium readable by a computer, tangibly embodying a program of instructions executable by the computer to perform method steps for intelligent server streaming of conventionally coded streamed application programs across a computer network while concurrently executing said streamed application programs on a client in a computer environment, comprising the steps of:

downloading an initial portion of a streamed application program on said client wherein said streamed application program comprises page segments and wherein said initial portion of said streamed application program remains on said client after terminating execution of said streamed application program by said client;

providing an application server;

partitioning said streamed application program into said page segments on said application server;

wherein said application server streams said page segments to said client upon said client's request;

wherein the user starts said streamed application program as if said streamed application program were fully installed on said client;

wherein specific page segments are requested by said client's file system during execution of said streamed application program such that said streamed application program begins execution of said client prior to downloading all of said page segments; and

storing said page segments in a cache on said client.

32. The method of claim 31, wherein said streamed application program is not recompiled, rewritten, or rebuilt for this specific delivery mechanism.

33. The method of claim 31, wherein said client manages said cache by purging page segments that are stale or not needed.

34. The method of claim 31, wherein said client does not request page segments of said streamed application program that already reside in said cache.

35. The method of claim 31, further comprising the step of:

providing a subscription server; and

wherein the user subscribes or unsubscribes to streamed application programs with said subscription server.

36. The method of claim 31, further comprising the step of:

providing a license server; and

wherein said client obtains an access token for a requested streamed application program from said license server if the user has a valid subscription to said requested streamed application program.

37. The method of claim 36, wherein said access token contains an expiration tag.

38. The method of claim 36, wherein said access token is securely encrypted.

39. The method of claim 36, wherein said client passes said access token to said application server before requesting page segments of said streamed application program.

40. The method of claim 36, wherein if said license server fails said client automatically switches to another license server.

41. The method of claim 31, further comprising the step of:

providing a profile information database characterizing the typical page segment needs of each streamed application program on said application server.

42. The method of claim 41, wherein said profile information database is updated dynamically as page segments are requested from said application server.

43. The method of claim 41, wherein said client prefetches page segments of said streamed application program from said application server based on the profile information of said streamed application program.

44. The method of claim 41, wherein said application server pushes page segments of said streamed application program to said client based on the profile information of said streamed application program.

45. The method of claim 31, wherein said client performs load balancing among a plurality of application servers for page segment requests.

Exhibit B

Content Management System (CMS) Files as of November 6, 2000

file	revision	action
//depot/docs/AppInstallManager-LLD.doc	1	add
//depot/docs/AppInstallManager-LLD.doc	2	edit
//depot/docs/AppInstallManager-LLD.doc	3	edit
//depot/docs/AppInstallManager-LLD.doc	4	edit
//depot/docs/Builder/AppInstallBlock-LLD.doc	1	add
//depot/docs/Builder/AppInstallBlock-LLD.doc	2	edit
//depot/docs/Builder/AppInstallBlock-LLD.doc	3	edit
//depot/docs/Builder/AppInstallBlock-LLD.doc	4	edit
//depot/docs/Builder/AppInstallBlock-LLD.doc	5	edit
//depot/docs/Builder/AppInstallBlock-LLD.doc	6	edit
//depot/docs/Builder/AppInstallBlock-LLD.doc	7	edit
//depot/docs/Builder/AppInstallBlock-LLD.doc	8	edit
//depot/docs/Builder/AppInstallBlock-LLD.doc	9	edit
//depot/docs/Builder/AppInstallBlock-LLD.doc	10	edit
//depot/docs/Builder/AppInstallBlock-LLD.doc	11	edit
//depot/docs/Builder/AppInstallBlock-LLD.doc	12	edit
//depot/docs/Builder/AppInstallBlock-LLD.doc	13	edit
//depot/docs/Builder/AppInstallBlock-format.vsd	1	add
//depot/docs/Builder/AppInstallBlock-format.vsd	2	edit
//depot/docs/Builder/AppInstallBlock-format.vsd	3	edit
//depot/docs/Builder/AppInstallBlock-format.vsd	4	edit
//depot/docs/Builder/AppInstallBlock-format.vsd	5	edit
//depot/docs/Builder/AppInstallBlock-format.vsd	6	edit
//depot/docs/Builder/AppInstallBlock-format.vsd	7	edit
//depot/docs/Builder/AppInstallBlock-format.vsd	8	edit
//depot/docs/Builder/Builder-HLD.doc	1	add
//depot/docs/Builder/Builder-HLD.doc	2	edit
//depot/docs/Builder/Builder-HLD.doc	3	edit
//depot/docs/Builder/Builder-HLD.doc	4	edit
//depot/docs/Builder/Builder-HLD.doc	5	edit
//depot/docs/Builder/Builder-HLD.doc	6	edit
//depot/docs/Builder/Builder-HLD.doc	7	edit
//depot/docs/Builder/Builder-HLD.doc	8	edit
//depot/docs/Builder/Builder-HLD.doc	9	edit
//depot/docs/Builder/Builder-HLD.doc	10	edit
//depot/docs/Builder/Builder-HLD.doc	11	edit
//depot/docs/Builder/Builder-HLD.doc	12	edit
//depot/docs/Builder/Builder-HLD.doc	13	edit
//depot/docs/Builder/Builder-LLD.doc	1	add
//depot/docs/Builder/Builder-LLD.doc	2	edit
//depot/docs/Builder/Builder-LLD.doc	3	edit
//depot/docs/Builder/BuilderTrickyIssues.doc	1	add
//depot/docs/Builder/BuilderUI-LLD.doc	1	add
//depot/docs/Builder/BuilderUI-LLD.doc	2	edit
//depot/docs/Builder/FSRFD-LLD.doc	1	add
//depot/docs/Builder/FSRFD-LLD.doc	2	edit
//depot/docs/Builder/FSRFD-LLD.doc	3	edit
//depot/docs/Builder/FSRFD-LLD.doc	4	edit
//depot/docs/Builder/Installmon-LLD.doc	1	add
//depot/docs/Builder/Installmon-LLD.doc	2	edit
//depot/docs/Builder/Installmon-LLD.doc	3	edit

//depot/docs/Builder/builder-desc.doc	1 add
//depot/docs/Builder/builder-desc.doc	2 edit
//depot/docs/Builder/builder-desc.vsd	1 add
//depot/docs/Builder/builder-diagram.vsd	1 add
//depot/docs/Builder/builder-diagram.vsd	2 edit
//depot/docs/Builder/builder-diagram.vsd	3 edit
//depot/docs/Builder/builder-diagram.vsd	4 edit
//depot/docs/Builder/builder-diagram.vsd	5 edit
//depot/docs/Builder/builder-fam-lld.doc	1 add
//depot/docs/Builder/builder-fam-lld.doc	2 edit
//depot/docs/Builder/builder-fam-lld.doc	3 edit
//depot/docs/Builder/builder-fam-lld.doc	4 edit
//depot/docs/Builder/builder-fam-lld.doc	5 edit
//depot/docs/Builder/builder-fam-lld.doc	6 edit
//depot/docs/Builder/builder-package-lld.doc	1 add
//depot/docs/Builder/builder-package-lld.doc	2 edit
//depot/docs/Builder/builder-package-lld.doc	3 edit
//depot/docs/Builder/builder-package-lld.doc	4 edit
//depot/docs/Builder/builder-profile-lld.doc	1 add
//depot/docs/Builder/builder-profile-lld.doc	2 edit
//depot/docs/Builder/builder-profile-lld.doc	3 edit
//depot/docs/Builder/builder-profile-lld.doc	4 edit
//depot/docs/Builder/builder-profile-lld.doc	5 edit
//depot/docs/Builder/builder-profile-lld.doc	6 edit
//depot/docs/Builder/builder-profile-lld.doc	7 edit
//depot/docs/Builder/builder-profile-lld.doc	8 edit
//depot/docs/Builder/builder-profile-lld.doc	9 edit
//depot/docs/Builder/eStreamSet-lld.doc	1 add
//depot/docs/Builder/eStreamSet-lld.doc	2 edit
//depot/docs/Builder/eStreamSet-lld.doc	3 edit
//depot/docs/Builder/eStreamSet-lld.doc	4 edit
//depot/docs/Builder/eStreamSet-lld.doc	5 edit
//depot/docs/Builder/eStreamSet-lld.doc	6 edit
//depot/docs/Builder/eStreamSet-lld.doc	7 edit
//depot/docs/Builder/eStreamSet-lld.doc	8 edit
//depot/docs/Builder/eStreamSet-lld.doc	9 edit
//depot/docs/Builder/eStreamSet-lld.doc	10 edit
//depot/docs/Builder/eStreamSet-lld.doc	11 edit
//depot/docs/Builder/eStreamSet-lld.doc	12 edit
//depot/docs/Builder/estream-set-format.vsd	1 add
//depot/docs/Builder/estream-set-format.vsd	2 delete
//depot/docs/Builder/estreamSet-format.vsd	1 add
//depot/docs/Builder/estreamSet-format.vsd	2 edit
//depot/docs/Builder/estreamSet-format.vsd	3 edit
//depot/docs/Builder/estreamSet-format.vsd	4 edit
//depot/docs/Builder/estreamSet-format.vsd	5 edit
//depot/docs/CUILLD.doc	1 add
//depot/docs/CUILLD.doc	2 delete
//depot/docs/CUIStrawMan.doc	1 add
//depot/docs/CUIStrawMan.doc	2 edit
//depot/docs/CUIStrawMan.doc	3 edit
//depot/docs/CUIStrawMan.doc	4 delete

//depot/docs/Client Docs/Client Spoof Registry.doc	1 add
//depot/docs/Client Docs/Client Spoof Registry.doc	2 edit
//depot/docs/Client Docs/Client UI Module.doc	1 add
//depot/docs/Client Docs/Client UI Module.doc	2 edit
//depot/docs/Client Docs/Client UI Module.doc	3 edit
//depot/docs/Client Docs/Installation Manager.doc	1 add
//depot/docs/Client Docs/Installation Manager.doc	2 edit
//depot/docs/Client Docs/License Subscription Manager (LSM).doc	1 add
//depot/docs/Client Docs/License Subscription Manager (LSM).doc	2 edit
//depot/docs/Client Docs/License Subscription Manager (LSM).doc	3 edit
//depot/docs/Client Docs/License Subscription Manager (LSM).doc	4 edit
//depot/docs/Client Docs/eStream Client Network Component.doc	1 add
//depot/docs/Client Docs/eStream Registry Spoofer current status.doc	1 add
//depot/docs/Client Install.doc	1 add
//depot/docs/ClientNW-LLD.doc	1 add
//depot/docs/ClientNW-LLD.doc	2 edit
//depot/docs/Configuration.vsd	1 add
//depot/docs/Downgrade.vsd	1 add
//depot/docs/ECMLLD.doc	1 add
//depot/docs/ECMLLD.doc	2 edit
//depot/docs/ECMLLD.doc	3 edit
//depot/docs/ECMLLD.doc	4 edit
//depot/docs/ECMStrawMan.doc	1 add
//depot/docs/Estream Client-Server Diagram.vsd	1 add
//depot/docs/Estream High-Level Design Diagram.vsd	1 add
//depot/docs/Estream High-Level Design Diagram.vsd	2 edit
//depot/docs/Install a subscribed application.doc	1 add
//depot/docs/Install a subscribed application.doc	2 edit
//depot/docs/Install a subscribed application.doc	3 edit
//depot/docs/Install.vsd	1 add
//depot/docs/LSMLLD.doc	1 add
//depot/docs/LSMLLD.doc	2 edit
//depot/docs/LSMLLD.doc	3 delete
//depot/docs/LSMStrawMan.doc	1 add
//depot/docs/LSMStrawMan.doc	2 edit
//depot/docs/LSMStrawMan.doc	3 edit
//depot/docs/LSMStrawMan.doc	4 delete
//depot/docs/License Subscription Manager (LSM).doc	1 add
//depot/docs/Omnishift BasePatent.doc	1 add
//depot/docs/Omnishift BasePatent.doc	2 edit
//depot/docs/Omnishift BasePatent.doc	3 edit
//depot/docs/Omnishift BasePatent.doc	4 edit
//depot/docs/Omnishift BasePatent.doc	5 edit
//depot/docs/Omnishift BasePatent.doc	6 edit
//depot/docs/OmnishiftCodingStandard.doc	1 add
//depot/docs/OmnishiftCodingStandard.doc	2 edit
//depot/docs/OmnishiftCodingStandard.doc	3 edit
//depot/docs/OmnishiftCodingStandard.doc	4 delete
//depot/docs/PrefetchStrawMan.doc	1 add
//depot/docs/PrefetchStrawMan.doc	2 delete
//depot/docs/QUESTIONS TO ASK AN ASP.doc	1 add
//depot/docs/QUESTIONS TO ASK AN ASP.doc	2 edit

//depot/docs/QUESTIONS TO ASK AN ASP.doc	3 edit
//depot/docs/QUESTIONS TO ASK AN ASP.doc	4 delete
//depot/docs/SLiM-LLD.doc	1 add
//depot/docs/SLiM-LLD.doc	2 delete
//depot/docs/ServerDocs/AppServer-LLD.doc	1 add
//depot/docs/ServerDocs/AppServer-LLD.doc	2 edit
//depot/docs/ServerDocs/AppServer-LLD.doc	3 edit
//depot/docs/ServerDocs/ComponentFramework-LLD.doc	1 add
//depot/docs/ServerDocs/ComponentFramework-LLD.doc	2 edit
//depot/docs/ServerDocs/ComponentFramework-LLD.doc	3 edit
//depot/docs/ServerDocs/ComponentFramework-LLD.doc	4 delete
//depot/docs/ServerDocs/Monitor-LLD.doc	1 add
//depot/docs/ServerDocs/Monitor-LLD.doc	2 edit
//depot/docs/ServerDocs/Monitor-LLD.doc	3 edit
//depot/docs/ServerDocs/Monitor-LLD.doc	4 edit
//depot/docs/ServerDocs/Monitor-LLD.doc	5 delete
//depot/docs/ServerDocs/SLiM-LLD.doc	1 add
//depot/docs/ServerDocs/SLiM-LLD.doc	2 edit
//depot/docs/ServerDocs/SLiM-LLD.doc	3 edit
//depot/docs/ServerDocs/SLiM-LLD.doc	4 edit
//depot/docs/ServerDocs/WebServerDB-LLD.doc	1 add
//depot/docs/ServerDocs/WebServerDB-LLD.doc	2 edit
//depot/docs/ServerDocs/WebServerDesign.doc	1 add
//depot/docs/ServerDocs/WebServerDesign.doc	2 edit
//depot/docs/ServerDocs/WebServerDesign.doc	3 delete
//depot/docs/ServerDocs/corba_centric.doc	1 add
//depot/docs/ServerDocs/db_centric.doc	1 add
//depot/docs/ServerDocs/eStreamConfigManagement.doc	1 add
//depot/docs/ServerDocs/eStreamConfigManagement.doc	2 edit
//depot/docs/ServerDocs/eStreamHTTP.doc	1 add
//depot/docs/ServerDocs/eStreamLoggingDesign.doc	1 add
//depot/docs/ServerDocs/eStreamLoggingDesign.doc	2 edit
//depot/docs/Uninstalling a subscribed application.doc	1 add
//depot/docs/Upgrade1.vsd	1 add
//depot/docs/Upgrade2.vsd	1 add
//depot/docs/Upgrade3.vsd	1 add
//depot/docs/client-driver-design.htm	1 add
//depot/docs/client-driver-design.htm	2 edit
//depot/docs/client-driver-design.htm	3 edit
//depot/docs/client_scenarios.htm	1 add
//depot/docs/eStream 1.0 Functional Spec.doc	1 add
//depot/docs/eStream 1.0 Functional Spec.doc	2 edit
//depot/docs/eStream 1.0 Functional Spec.doc	3 edit
//depot/docs/eStream 1.0 Functional Spec.doc	4 edit
//depot/docs/eStream 1.0 Functional Spec.doc	5 edit
//depot/docs/eStream 1.0 Requirements.doc	1 add
//depot/docs/eStream 1.0 Requirements.doc	2 edit
//depot/docs/eStream 1.0 Requirements.doc	3 edit
//depot/docs/eStream 1.0 Requirements.doc	4 edit
//depot/docs/eStream 1.0 Requirements.doc	5 delete
//depot/docs/eStream 1.0 Top Level Components.doc	1 add
//depot/docs/eStream1.0-HLD.doc	1 add

//depot/docs/eStream1.0-HLD.doc	2 delete
//depot/docs/eStream1.0-REQ.doc	1 branch
//depot/docs/eStream1.0-REQ.doc	2 edit
//depot/docs/eStream1.0-REQ.doc	3 edit
//depot/docs/eStream1.0-REQ.doc	4 edit
//depot/docs/eStream1.0-REQ.doc	5 edit
//depot/docs/eStream1.0-REQ.doc	6 edit
//depot/docs/eStream1.0-REQ.doc	7 edit
//depot/docs/eStream1.0-REQ.doc	8 delete
//depot/docs/eStream1.0-SCALE.doc	1 add
//depot/docs/eStream1.0-SCALE.doc	2 edit
//depot/docs/eStream1.0-SCALE.doc	3 delete
//depot/docs/eStreamStrawMan.doc	1 add
//depot/docs/eStreamStrawMan.doc	2 edit
//depot/docs/efsd-lld.doc	1 add
//depot/docs/efsd-lld.doc	2 edit
//depot/docs/efsd-lld.doc	3 edit
//depot/docs/efsd-lld.doc	4 edit
//depot/docs/file-spoofer-lld.doc	1 add
//depot/docs/osrdocs/apc.html	1 add
//depot/docs/osrdocs/blue-screen.html	1 add
//depot/docs/osrdocs/blue-screen_files/Image13.gif	1 add
//depot/docs/osrdocs/c++-support.html	1 add
//depot/docs/osrdocs/cacheman.html	1 add
//depot/docs/osrdocs/cancel-1.html	1 add
//depot/docs/osrdocs/cancel-2.html	1 add
//depot/docs/osrdocs/converting-nt4-5.html	1 add
//depot/docs/osrdocs/defensive-driv.html	1 add
//depot/docs/osrdocs/driver101.html	1 add
//depot/docs/osrdocs/driver101_files/arch.gif	1 add
//depot/docs/osrdocs/driver101_files/setup.gif	1 add
//depot/docs/osrdocs/driver102.html	1 add
//depot/docs/osrdocs/fs-recognizers.html	1 add
//depot/docs/osrdocs/impl-pnp.html	1 add
//depot/docs/osrdocs/integ-build.html	1 add
//depot/docs/osrdocs/io-completion.html	1 add
//depot/docs/osrdocs/io-completion_files/iocomp1.gif	1 add
//depot/docs/osrdocs/io-completion_files/iocomp2.gif	1 add
//depot/docs/osrdocs/io-completion_files/iocomp3.gif	1 add
//depot/docs/osrdocs/kernel-context.html	1 add
//depot/docs/osrdocs/kernel-dlls.html	1 add
//depot/docs/osrdocs/lanman-server.html	1 add
//depot/docs/osrdocs/nt-io-requests.html	1 add
//depot/docs/osrdocs/oplocks.html	1 add
//depot/docs/osrdocs/rename.html	1 add
//depot/docs/osrdocs/security1.html	1 add
//depot/docs/osrdocs/seh.html	1 add
//depot/docs/osrdocs/sync-mech.html	1 add
//depot/docs/osrdocs/vm-1.html	1 add
//depot/docs/osrdocs/vm-2.html	1 add
//depot/docs/osrdocs/windbg-ext.html	1 add
//depot/docs/osrdocs/wmi.html	1 add

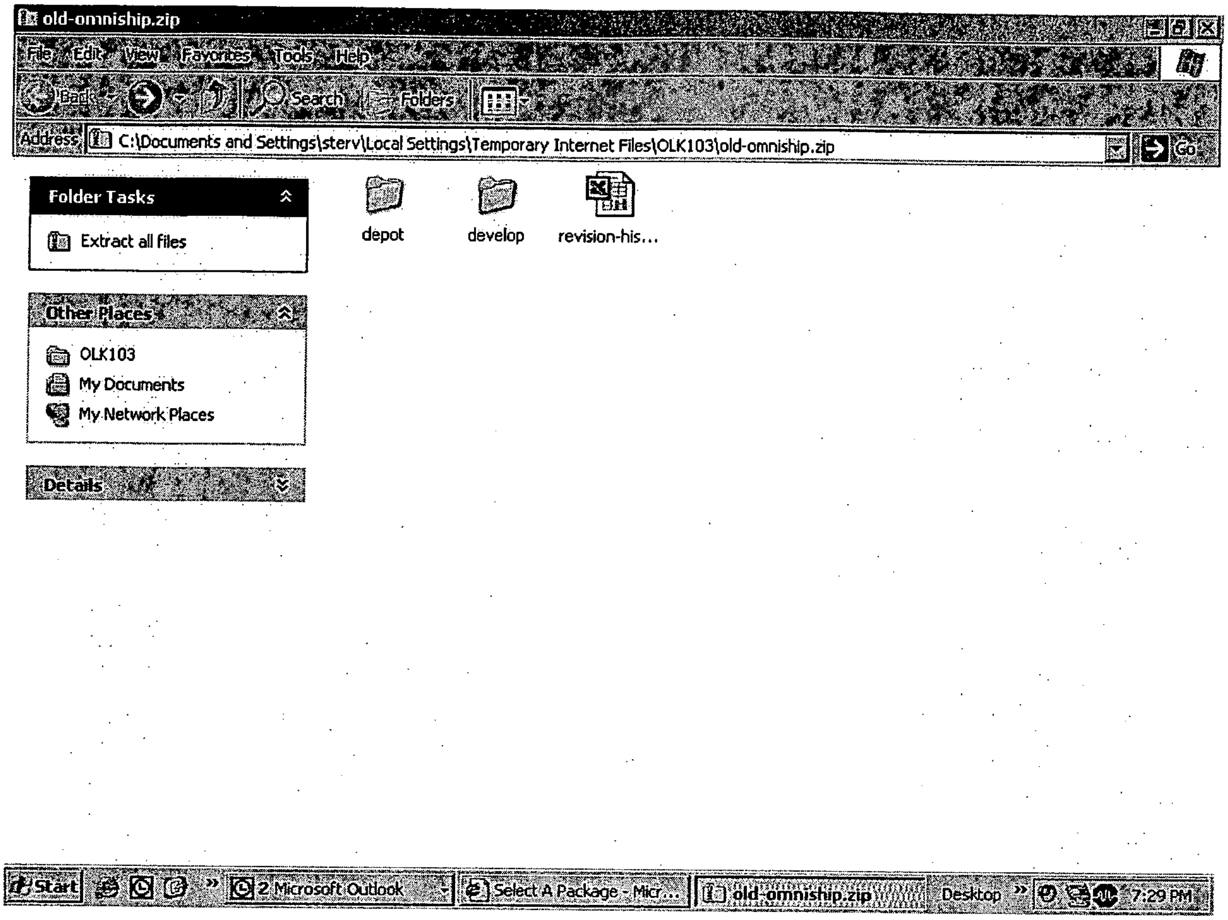
//depot/docs/osrdocs/work-queues.html	1 add
//develop/docs/Builder/AppInstallBlock-LLD.doc	1 add
//develop/docs/Builder/AppInstallBlock-LLD.doc	2 edit
//develop/docs/Builder/AppInstallBlock-LLD.doc	3 edit
//develop/docs/Builder/AppInstallBlock-LLD.doc	4 edit
//develop/docs/Builder/AppInstallBlock-LLD.doc	5 edit
//develop/docs/Builder/AppInstallBlock-format.vsd	1 add
//develop/docs/Builder/Builder-HLD.doc	1 add
//develop/docs/Builder/Builder-LLD.doc	1 add
//develop/docs/Builder/FSRFD-LLD.doc	1 add
//develop/docs/Builder/Installmon-LLD.doc	1 add
//develop/docs/Builder/TrickyBuilderIssues.doc	1 add
//develop/docs/Builder/builder-desc.doc	1 add
//develop/docs/Builder/builder-desc.vsd	1 add
//develop/docs/Builder/builder-diagram.vsd	1 add
//develop/docs/Builder/builder-fam-lld.doc	1 add
//develop/docs/Builder/builder-fam-lld.doc	2 edit
//develop/docs/Builder/builder-fam-lld.doc	3 edit
//develop/docs/Builder/builder-package-lld.doc	1 add
//develop/docs/Builder/builder-package-lld.doc	2 edit
//develop/docs/Builder/builder-package-lld.doc	3 edit
//develop/docs/Builder/builder-profile-lld.doc	1 add
//develop/docs/Builder/builder-profile-lld.doc	2 edit
//develop/docs/Builder/builder-profile-lld.doc	3 edit
//develop/docs/Builder/eStreamSet-lld.doc	1 add
//develop/docs/Builder/eStreamSet-lld.doc	2 edit
//develop/docs/Builder/eStreamSet-lld.doc	3 edit
//develop/docs/Builder/estreamSet-format.vsd	1 add
//develop/docs/Client Docs/Client Spoof Registry.doc	1 add
//develop/docs/Client Docs/Client UI Module.doc	1 add
//develop/docs/Client Docs/Installation Manager.doc	1 add
//develop/docs/Client Docs/License Subscription Manager (LSM).doc	1 add
//develop/docs/Client Docs/eStream Client Network Component.doc	1 add
//develop/docs/Client Docs/eStream Registry Spoofer current status.doc	1 add
//develop/docs/Client Install.doc	1 add
//develop/docs/Configuration.vsd	1 add
//develop/docs/Downgrade.vsd	1 add
//develop/docs/ECMLLD.doc	1 add
//develop/docs/ECMStrawMan.doc	1 add
//develop/docs/Estream Client-Server Diagram.vsd	1 add
//develop/docs/Estream High-Level Design Diagram.vsd	1 add
//develop/docs/Install a subscribed application.doc	1 add
//develop/docs/Install.vsd	1 add
//develop/docs/LSMLLD.doc	1 add
//develop/docs/License Subscription Manager (LSM).doc	1 add
//develop/docs/Omnishift BasePatent.doc	1 add
//develop/docs/OmnishiftCodingStandard.doc	1 add
//develop/docs/QUESTIONS TO ASK AN ASP.doc	1 add
//develop/docs/ServerDocs/ComponentFramework-LLD.doc	1 add
//develop/docs/ServerDocs/ComponentFramework-LLD.doc	2 edit
//develop/docs/ServerDocs/ComponentFramework-LLD.doc	3 edit
//develop/docs/ServerDocs/ComponentFramework-LLD.doc	4 edit

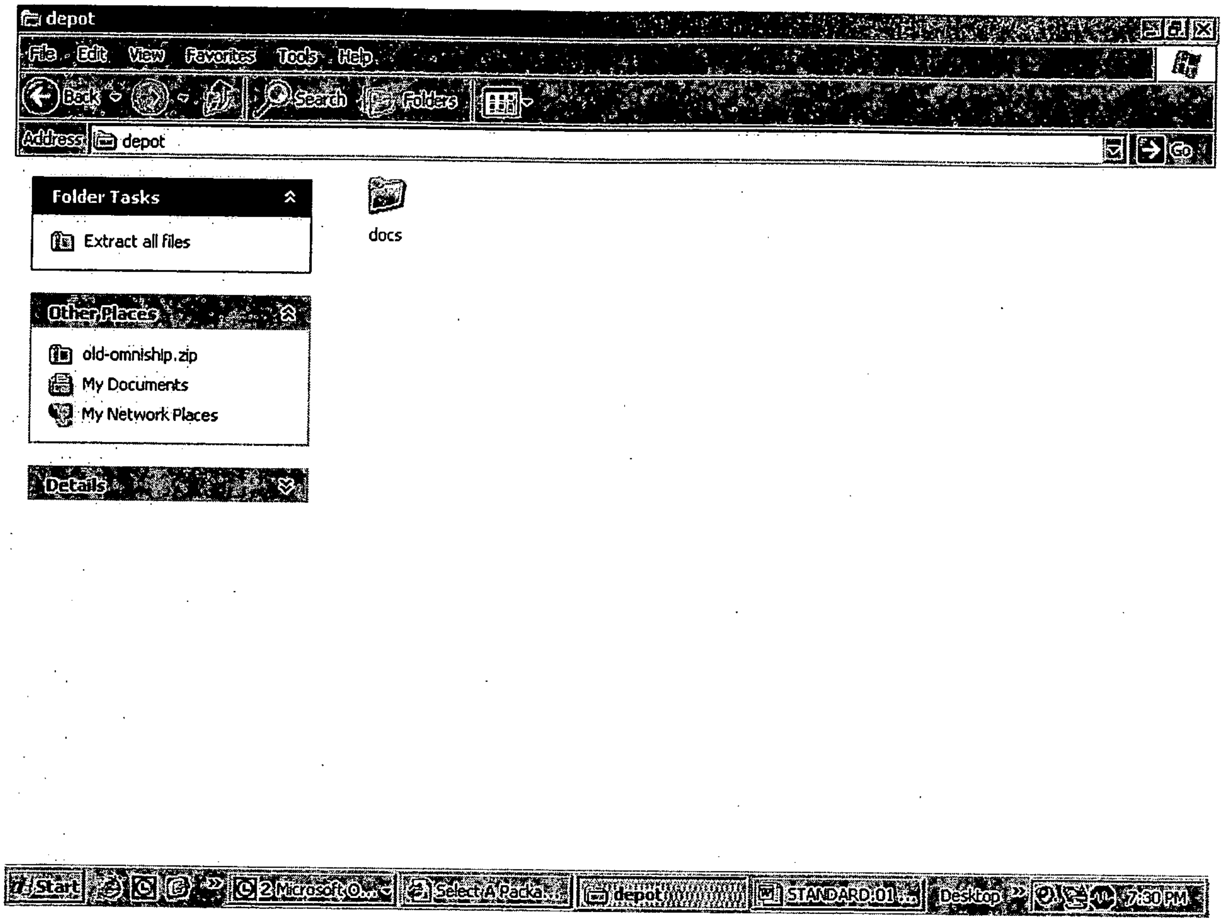
//develop/docs/ServerDocs/ComponentFramework-LLD.doc	5 edit
//develop/docs/ServerDocs/Monitor-LLD.doc	1 add
//develop/docs/ServerDocs/Monitor-LLD.doc	2 edit
//develop/docs/ServerDocs/Monitor-LLD.doc	3 edit
//develop/docs/ServerDocs/SLiM-LLD.doc	1 add
//develop/docs/ServerDocs/SLiM-LLD.doc	2 edit
//develop/docs/ServerDocs/WebServerDesign.doc	1 add
//develop/docs/ServerDocs/corba_centric.doc	1 add
//develop/docs/ServerDocs/db_centric.doc	1 add
//develop/docs/ServerDocs/eStreamConfigManagement.doc	1 add
//develop/docs/ServerDocs/eStreamHTTP.doc	1 add
//develop/docs/ServerDocs/eStreamLoggingDesign.doc	1 add
//develop/docs/Uninstalling a subscribed application.doc	1 add
//develop/docs/Upgrade1.vsd	1 add
//develop/docs/Upgrade2.vsd	1 add
//develop/docs/Upgrade3.vsd	1 add
//develop/docs/client-driver-design.htm	1 add
//develop/docs/client_scenarios.htm	1 add
//develop/docs/eStream 1.0 Functional Spec.doc	1 add
//develop/docs/eStream 1.0 Top Level Components.doc	1 add
//develop/docs/eStream1.0-HLD.doc	1 add
//develop/docs/eStream1.0-REQ.doc	1 add
//develop/docs/eStream1.0-SCALE.doc	1 add
//develop/docs/eStreamStrawMan.doc	1 add
//develop/docs/efsd-lld.doc	1 add
//develop/docs/file-spoofer-lld.doc	1 add
//develop/docs/osrdocs/apc.html	1 add
//develop/docs/osrdocs/blue-screen.html	1 add
//develop/docs/osrdocs/blue-screen_files/Image13.gif	1 add
//develop/docs/osrdocs/c++-support.html	1 add
//develop/docs/osrdocs/cacheman.html	1 add
//develop/docs/osrdocs/cancel-1.html	1 add
//develop/docs/osrdocs/cancel-2.html	1 add
//develop/docs/osrdocs/converting-nt4-5.html	1 add
//develop/docs/osrdocs/defensive-driv.html	1 add
//develop/docs/osrdocs/driver101.html	1 add
//develop/docs/osrdocs/driver101_files/arch.gif	1 add
//develop/docs/osrdocs/driver101_files/setup.gif	1 add
//develop/docs/osrdocs/driver102.html	1 add
//develop/docs/osrdocs/fs-recognizers.html	1 add
//develop/docs/osrdocs/impl-pnp.html	1 add
//develop/docs/osrdocs/integ-build.html	1 add
//develop/docs/osrdocs/io-completion.html	1 add
//develop/docs/osrdocs/io-completion_files/iocomp1.gif	1 add
//develop/docs/osrdocs/io-completion_files/iocomp2.gif	1 add
//develop/docs/osrdocs/io-completion_files/iocomp3.gif	1 add
//develop/docs/osrdocs/kernel-context.html	1 add
//develop/docs/osrdocs/kernel-dlls.html	1 add
//develop/docs/osrdocs/lanman-server.html	1 add
//develop/docs/osrdocs/nt-io-requests.html	1 add
//develop/docs/osrdocs/oplocks.html	1 add
//develop/docs/osrdocs/rename.html	1 add

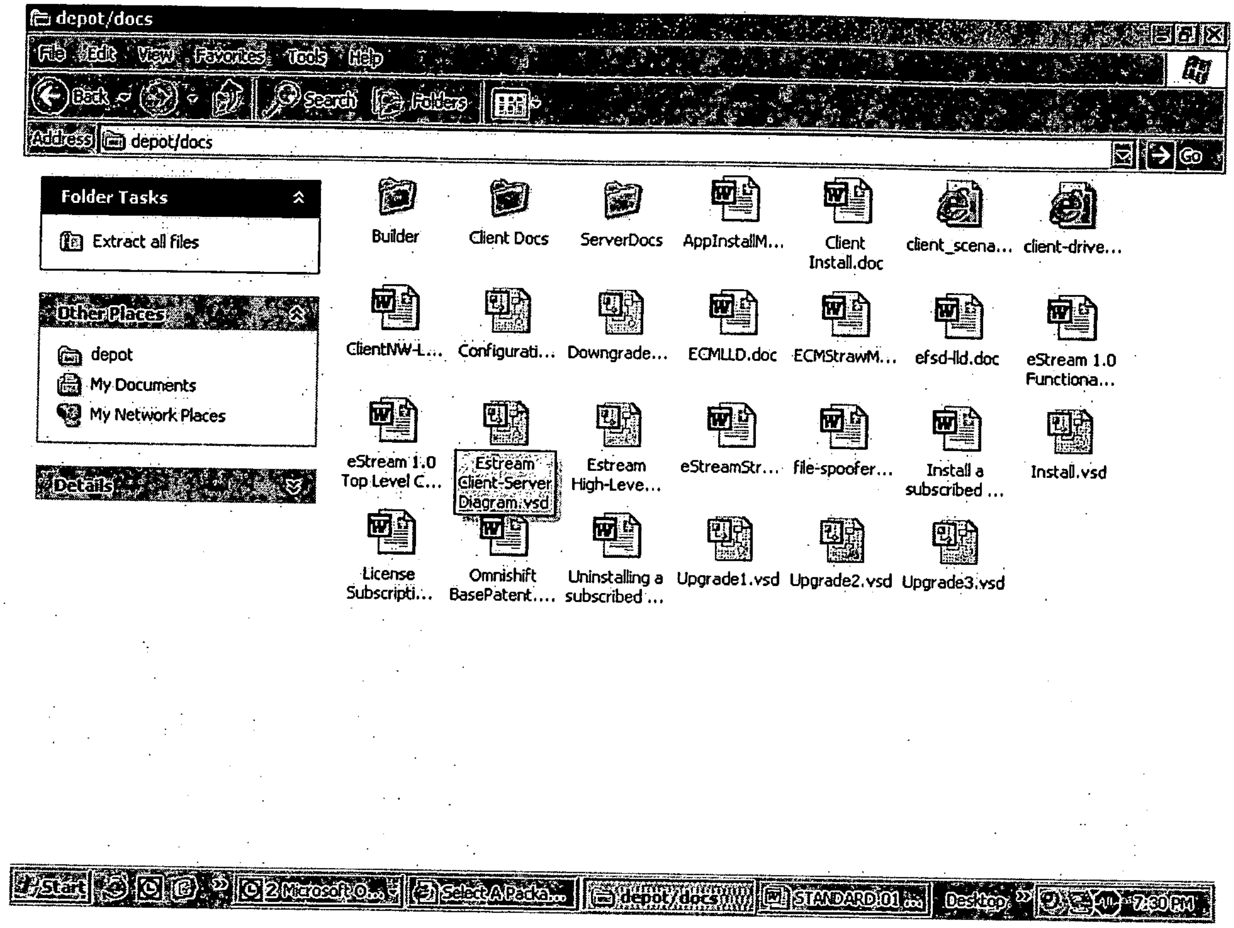
//develop/docs/osrdocs/security1.html	1 add
//develop/docs/osrdocs/seh.html	1 add
//develop/docs/osrdocs/sync-mech.html	1 add
//develop/docs/osrdocs/vm-1.html	1 add
//develop/docs/osrdocs/vm-2.html	1 add
//develop/docs/osrdocs/windbg-ext.html	1 add
//develop/docs/osrdocs/wmi.html	1 add
//develop/docs/osrdocs/work-queues.html	1 add
//develop/eng/docs/OmnishiftCodingStandard.doc	1 add
//develop/eng/docs/OmnishiftCodingStandard.doc	2 edit
//develop/eng/docs/OmnishiftCodingStandard.doc	3 edit
//develop/eng/docs/estream1.0-plan.xls	1 add
//develop/eng/docs/estream1.0-plan.xls	2 delete
//develop/eng/docs/readme.txt	1 add
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.doc	1 add
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.doc	2 edit
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.doc	3 edit
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.doc	4 edit
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.doc	5 edit
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.vsd	1 add
//develop/eng/estream/docs/builder/AppInstallBlock-LLD.vsd	2 edit
//develop/eng/estream/docs/builder/Builder-HLD.doc	1 add
//develop/eng/estream/docs/builder/Builder-HLD.doc	2 edit
//develop/eng/estream/docs/builder/Builder-HLD.vsd	1 add
//develop/eng/estream/docs/builder/Builder-HLD.vsd	2 edit
//develop/eng/estream/docs/builder/BuilderDesc-HLD.doc	1 add
//develop/eng/estream/docs/builder/BuilderDesc-HLD.doc	2 edit
//develop/eng/estream/docs/builder/BuilderDesc-HLD.vsd	1 add
//develop/eng/estream/docs/builder/BuilderDesc-HLD.vsd	2 edit
//develop/eng/estream/docs/builder/BuilderTrickyIssues.doc	1 add
//develop/eng/estream/docs/builder/EStreamSet-LLD.doc	1 add
//develop/eng/estream/docs/builder/EStreamSet-LLD.doc	2 edit
//develop/eng/estream/docs/builder/EStreamSet-LLD.doc	3 edit
//develop/eng/estream/docs/builder/EStreamSet-LLD.doc	4 edit
//develop/eng/estream/docs/builder/EStreamSet-LLD.doc	5 edit
//develop/eng/estream/docs/builder/EStreamSet-LLD.vsd	1 add
//develop/eng/estream/docs/builder/EStreamSet-LLD.vsd	2 edit
//develop/eng/estream/docs/builder/builderUI/BuilderUI-LLD.doc	1 add
//develop/eng/estream/docs/builder/fsrfd/FSRFD-LLD.doc	1 add
//develop/eng/estream/docs/builder/installmon/Installmon-LLD.doc	1 add
//develop/eng/estream/docs/builder/packager/BuilderPackager-LLD.doc	1 add
//develop/eng/estream/docs/builder/packager/BuilderPackager-LLD.doc	2 edit
//develop/eng/estream/docs/builder/packager/BuilderPackager-LLD.doc	3 edit
//develop/eng/estream/docs/builder/profiler/BuilderFAM-LLD.doc	1 add
//develop/eng/estream/docs/builder/profiler/BuilderFAM-LLD.doc	2 edit
//develop/eng/estream/docs/builder/profiler/BuilderProfiler-LLD.doc	1 add
//develop/eng/estream/docs/builder/profiler/BuilderProfiler-LLD.doc	2 edit
//develop/eng/estream/docs/builder/profiler/BuilderProfiler-LLD.doc	3 edit
//develop/eng/estream/docs/client/aim/AppInstallManager-LLD.doc	1 add
//develop/eng/estream/docs/client/aim/AppInstallManager-LLD.doc	2 edit
//develop/eng/estream/docs/client/aim/AppInstallManager-LLD.doc	3 edit
//develop/eng/estream/docs/client/aim/AppInstallManager-LLD.doc	4 edit

//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	1 add
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	2 edit
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	3 edit
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	4 edit
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	5 edit
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-LLD.doc	6 edit
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-SM.doc	1 add
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-SM.doc	2 edit
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-SM.doc	3 edit
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-SM.doc	4 edit
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-SM.doc	5 edit
//develop/eng/estream/docs/client/ces/ClientEstreamStartup-SM.doc	6 edit
//develop/eng/estream/docs/client/cni/ClientNetworking-LLD.doc	1 add
//develop/eng/estream/docs/client/cni/ClientNetworking-LLD.vsd	1 add
//develop/eng/estream/docs/client/cui/ClientUserInterface-LLD.doc	1 branch
//develop/eng/estream/docs/client/cui/ClientUserInterface-LLD.doc	2 edit
//develop/eng/estream/docs/client/cui/ClientUserInterface-LLD.doc	3 edit
//develop/eng/estream/docs/client/cui/ClientUserInterface-LLD.doc	4 edit
//develop/eng/estream/docs/client/cui/ClientUserInterface-LLD.doc	5 edit
//develop/eng/estream/docs/client/cui/ClientUserInterface-SM.doc	1 branch
//develop/eng/estream/docs/client/ecm/CacheManager-LLD.doc	1 add
//develop/eng/estream/docs/client/ecm/CacheManager-LLD.vsd	1 add
//develop/eng/estream/docs/client/ecm/CacheManager-SM.doc	1 add
//develop/eng/estream/docs/client/efsd/efsd-lld.doc	1 add
//develop/eng/estream/docs/client/epf/PrefetcherFetcher-LLD.doc	1 add
//develop/eng/estream/docs/client/epf/PrefetcherFetcher-LLD.doc	2 edit
//develop/eng/estream/docs/client/epf/PrefetcherFetcher-SM.doc	1 branch
//develop/eng/estream/docs/client/fsp/file-spoofer-lld.doc	1 add
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	1 branch
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	2 edit
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	3 edit
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	4 edit
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	5 edit
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	6 edit
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	7 edit
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	8 edit
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	9 edit
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-LLD.doc	10 edit
//develop/eng/estream/docs/client/lsm/LicenseSubscriptionMgr-SM.doc	1 branch
//develop/eng/estream/docs/development-plan.xls.xls	1 add
//develop/eng/estream/docs/eStream1.0-ASP.doc	1 branch
//develop/eng/estream/docs/eStream1.0-ASP.doc	2 delete
//develop/eng/estream/docs/eStream1.0-HLD.doc	1 branch
//develop/eng/estream/docs/eStream1.0-Req.doc	1 branch
//develop/eng/estream/docs/eStream1.0-SCALE.doc	1 branch
//develop/eng/estream/docs/eStreamFS-SM.doc	1 add
//develop/eng/estream/docs/eng/OmnishiftCodingStandard.doc	1 branch
//develop/eng/estream/docs/eng/OmnishiftCodingStandard.doc	2 delete
//develop/eng/estream/docs/mpr.doc	1 add
//develop/eng/estream/docs/readme.txt	1 add
//develop/eng/estream/docs/readme.txt	2 edit
//develop/eng/estream/docs/server/AppServer-LLD.doc	1 add

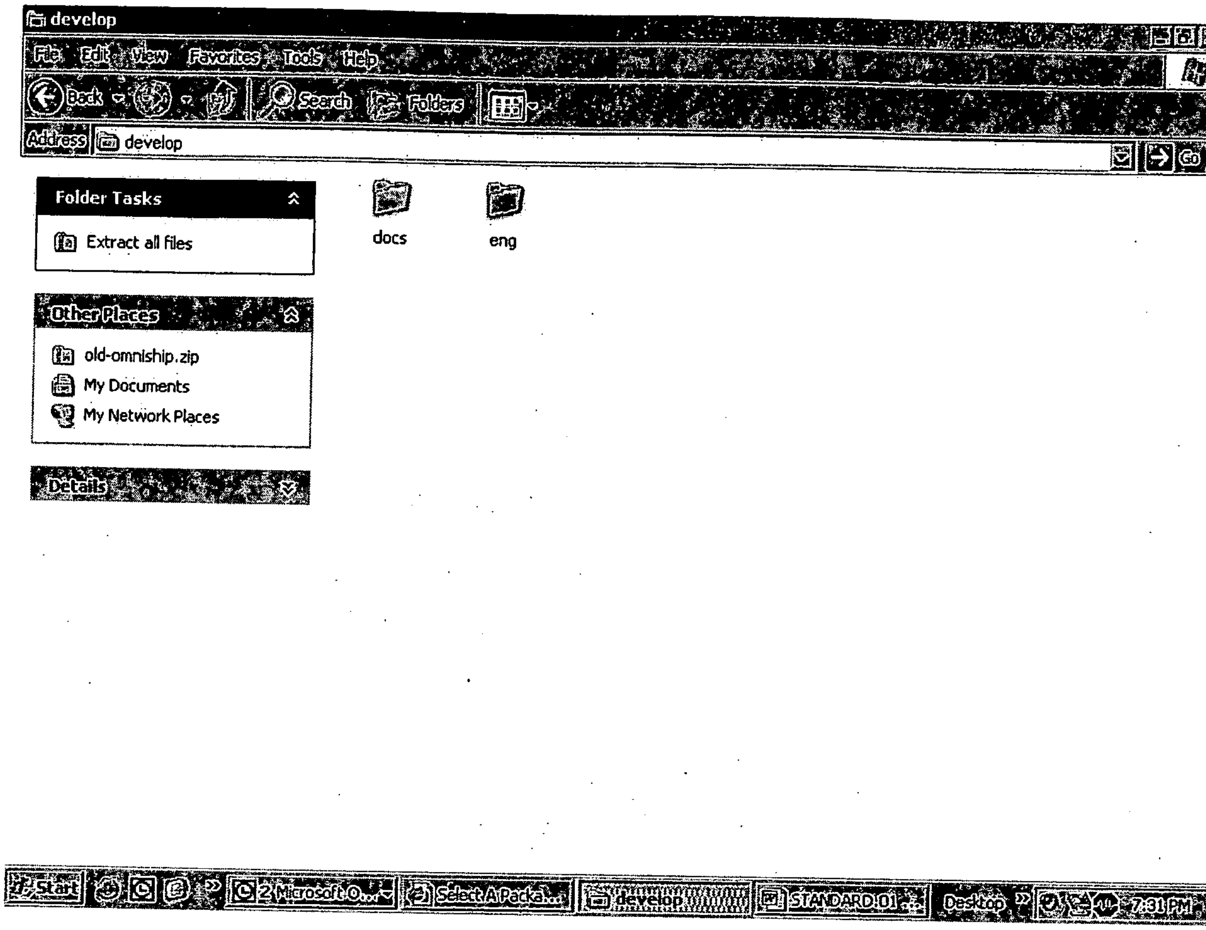
//develop/eng/estream/docs/server/ComponentFramework-LLD.doc	1 branch
//develop/eng/estream/docs/server/EMS-LLD.doc	1 add
//develop/eng/estream/docs/server/EMS-LLD.doc	2 edit
//develop/eng/estream/docs/server/EMS-LLD.doc	3 edit
//develop/eng/estream/docs/server/Monitor-LLD.doc	1 branch
//develop/eng/estream/docs/server/Monitor-LLD.doc	2 edit
//develop/eng/estream/docs/server/SLiM-LLD.doc	1 add
//develop/eng/estream/docs/server/WebServerDB-LLD.doc	1 add
//develop/eng/estream/docs/wbs.mpp	1 add

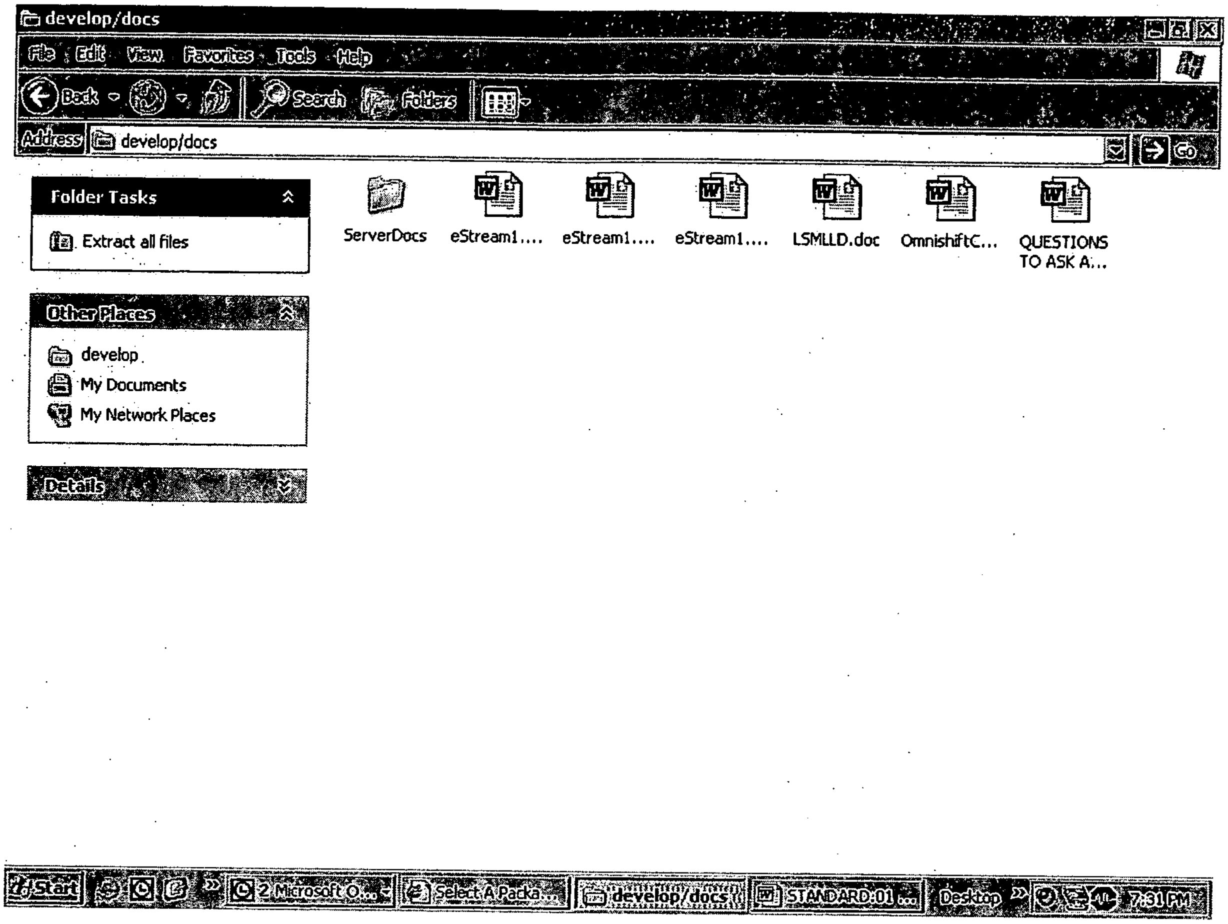

old-omniship.zip
File Edit View Favorites Tools Help
Back Search Folders
Address C:\Documents and Settings\sterv\Local Settings\Temporary Internet Files\OLK103\old-omniship.zip
Go
Folder Tasks
Extract all files
Other Places
OLK103
My Documents
My Network Places
Details
depot
develop
revision-his...
Start
2 Microsoft Outlook
Select A Package - Micr...
old-omniship.zip
Desktop
7:29 PM


depot
File Edit View Favorites Tools Help
Back
Search
Folders
Address
depot
Go
Folder Tasks
Extract all files
docs
Other Places
old-omniship.zip
My Documents
My Network Places
Details
Start
2 Microsoft O...
Select A Packa...
depot
STANDARD.01 ...
Desktop
7:30 PM

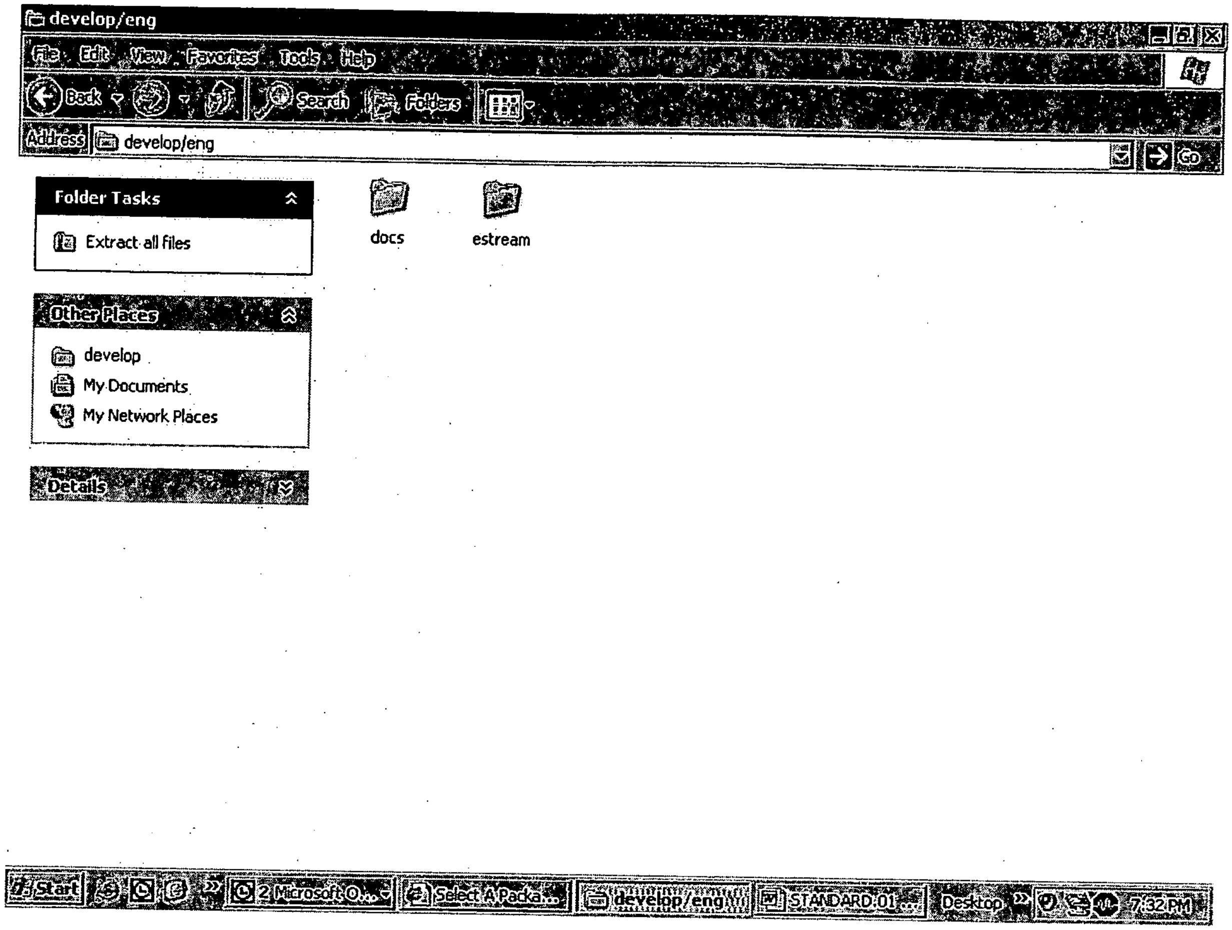

depot/docs
File Edit View Favorites Tools Help
Back Search Folders
Address depot/docs
Go
Folder Tasks
Extract all files
Other Places
depot
My Documents
My Network Places
Details
Builder
Client Docs
ServerDocs
AppInstallM...
Client Install.doc
client_scena...
client-drive...
ClientNW-L...
Configurati...
Downgrade...
ECMLLD.doc
ECMStrawM...
efsd-lld.doc
eStream 1.0 Functiona...
eStream 1.0 Top Level C...
Estream Client-Server Diagram.vsd
Estream High-Leve...
eStreamStr...
file-spoofer...
Install a subscribed ...
Install.vsd
License Subscripti...
Omnishift BasePatent....
Uninstalling a subscribed ...
Upgrade1.vsd
Upgrade2.vsd
Upgrade3.vsd
Start
2 Microsoft O...
Select A Packa...
depot/docs
STANDARD.01 ...
Desktop
7:30 PM

[/STANDARD.01]

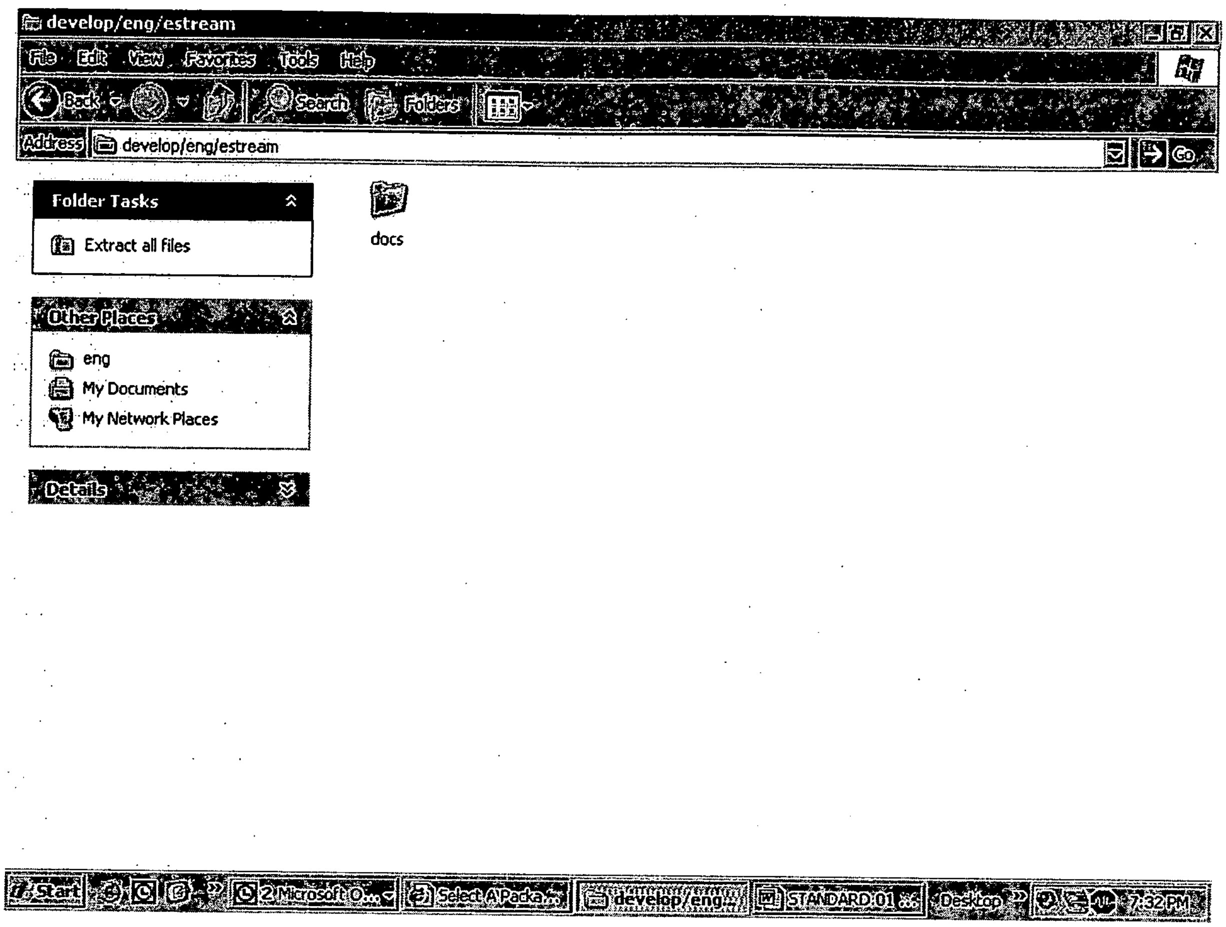

develop
File Edit View Favorites Tools Help
Back Search Folders
Address develop
Go
Folder Tasks
Extract all files
Other Places
old-omniship.zip
My Documents
My Network Places
Details
docs
eng
Start
Select A Packa...
develop
STANDARD.01
Desktop
7:31 PM


develop/docs
File Edit View Favorites Tools Help
Back Search Folders
Address develop/docs
Go
Folder Tasks
Extract all files
Other Places
develop
My Documents
My Network Places
Details
ServerDocs
eStream1....
eStream1....
eStream1....
LSMLLD.doc
OmnishiftC...
QUESTIONS TO ASK A...
Start
2 Microsoft O...
Select A Packa...
develop/docs
STANDARD.01 ...
Desktop
7:31 PM

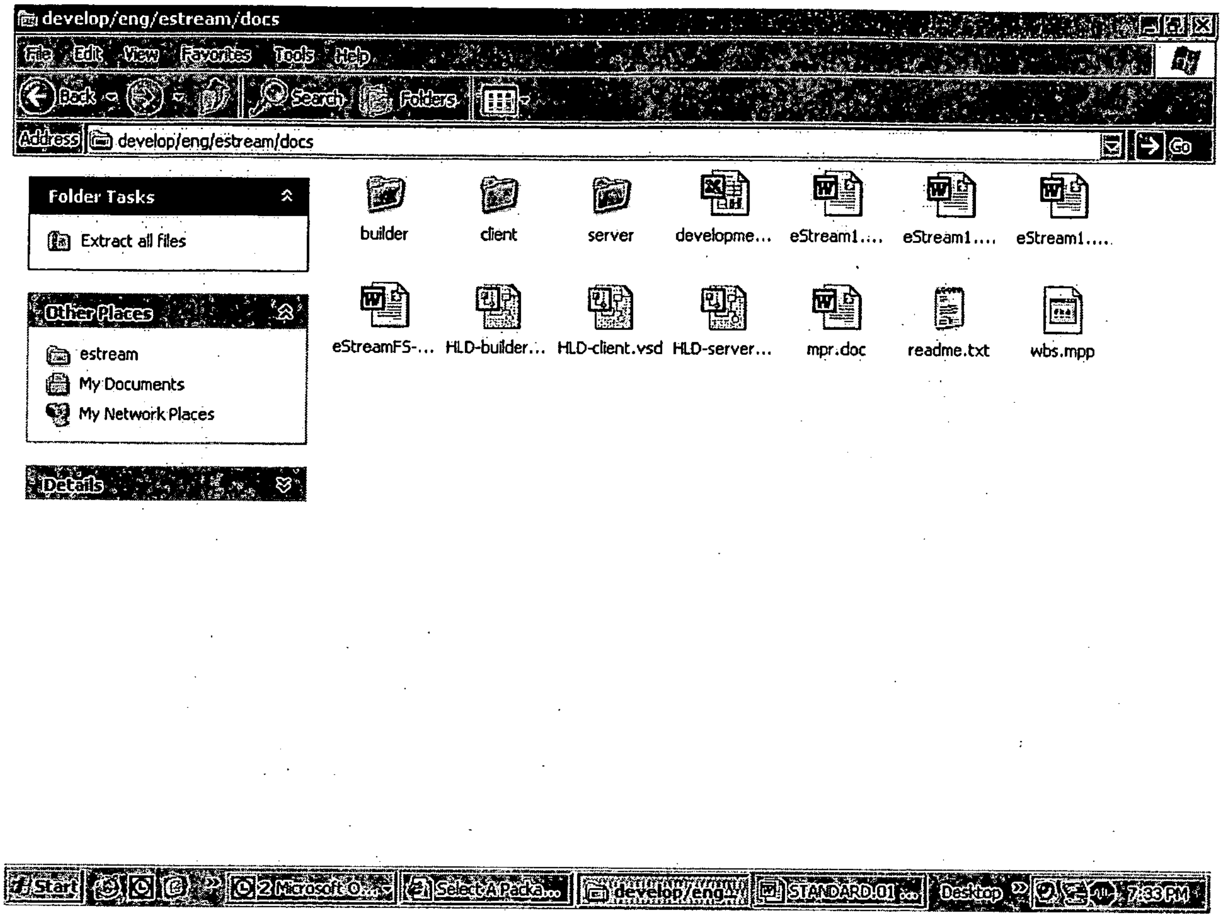
[/STANDARD.01]

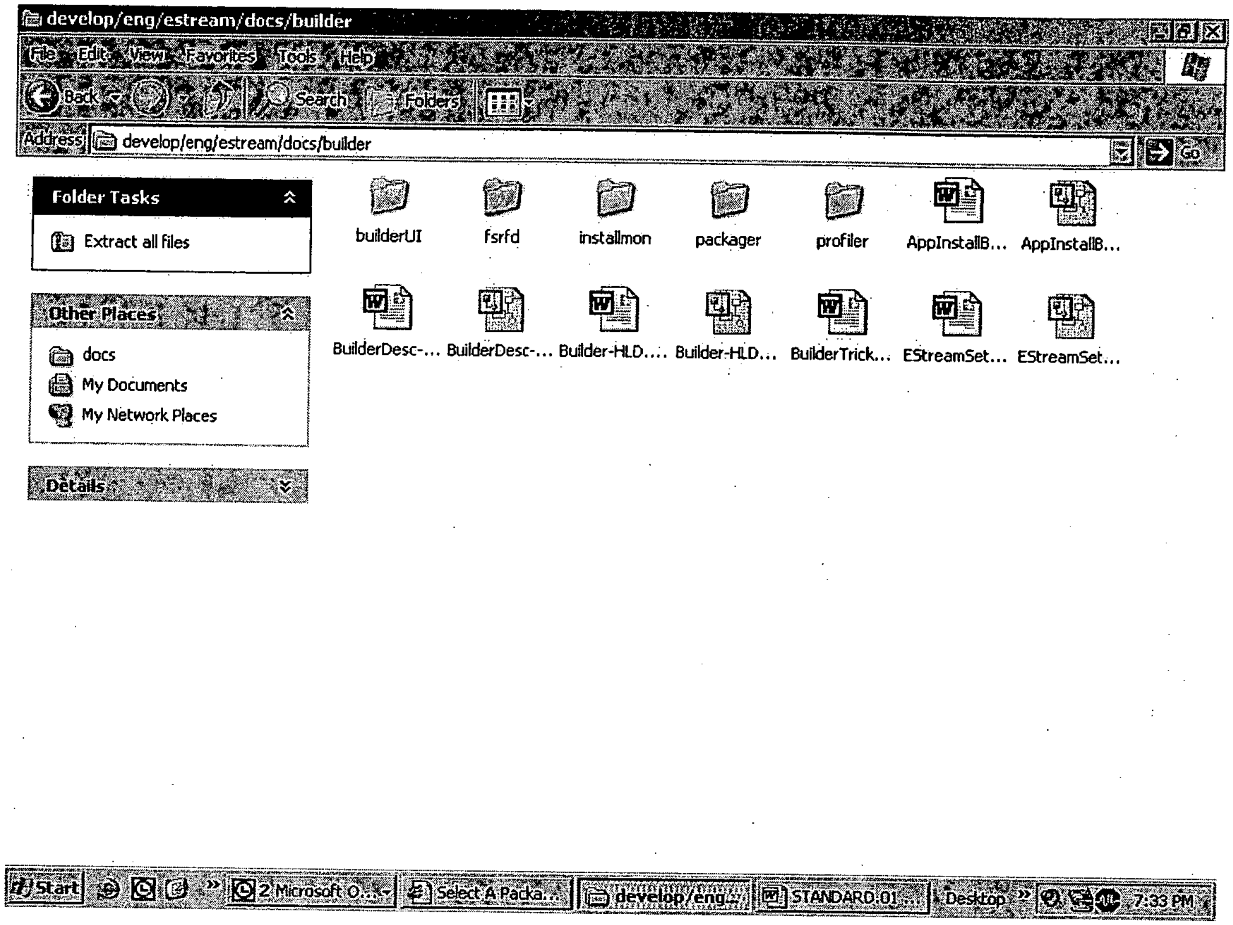

develop/eng
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng
Folder Tasks
Extract all files
Other Places
develop
My Documents
My Network Places
Details
docs
estream
Start
2 Microsoft O...
Select A Packa...
develop/eng
STANDARD.01...
Desktop
7:32 PM

[redacted]

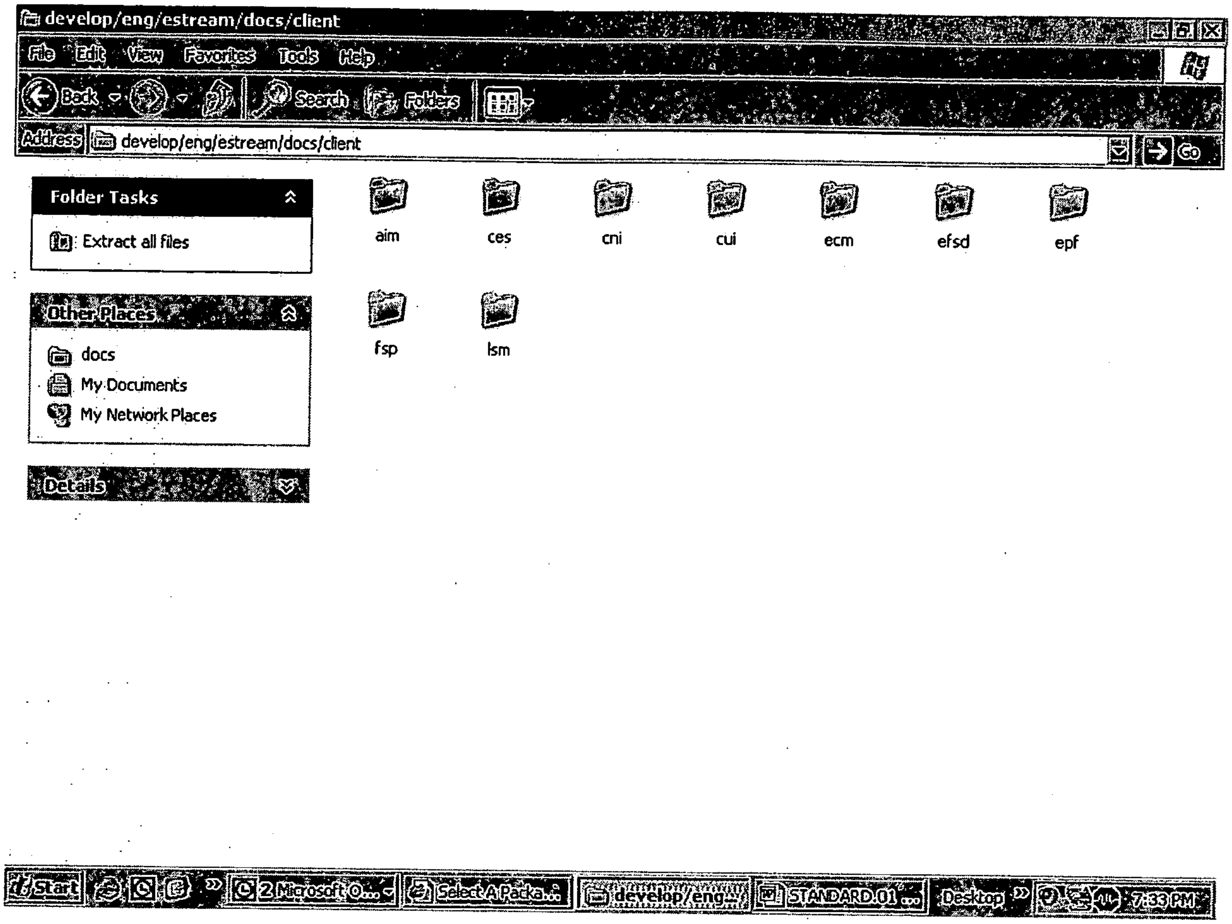

develop/eng/estream
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng/estream Go
Folder Tasks
Extract all files
docs
Other Places
eng
My Documents
My Network Places
Details
Start
2 Microsoft O...
Select A Packa...
develop/eng...
STANDARD.01 ...
Desktop
7:32 PM

[/STANDARD.01]


develop/eng/estream/docs
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng/estream/docs
Go
Folder Tasks
Extract all files
Other Places
estream
My Documents
My Network Places
Details
builder
client
server
developme...
eStream1....
eStream1....
eStream1....
eStreamFS-...
HLD-builder...
HLD-client.vsd
HLD-server...
mpr.doc
readme.txt
wbs.mpp
Start
2 Microsoft O...
Select A Packa...
develop/eng...
STANDARD.01...
Desktop
7:33 PM


develop/eng/estream/docs/builder
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng/estream/docs/builder
Go
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
builderUI
fsrfd
installmon
packager
profiler
AppInstallB...
AppInstallB...
BuilderDesc-...
BuilderDesc-...
Builder-HLD...
Builder-HLD...
BuilderTrick...
EStreamSet...
EStreamSet...
Start
2 Microsoft O...
Select A Packa...
develop/eng...
STANDARD.01...
Desktop
7:33 PM

[/STANDARD.01]


develop/eng/estream/docs/client
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng/estream/docs/client
Go
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm
Start
2 Microsoft O...
Select A Packa...
develop/eng...
STANDARD.01 ...
Desktop
7:33 PM

eStream Application Install Manager Low Level Design

Nicholas Ryan
Version 0.8
[redacted]

Functionality

The Application Install Manager (AIM) is a component of the eStream client executable. It is responsible for installing and uninstalling eStream applications at the request of the License Subscription Manager (LSM). AIM uses the information contained in an AppInstallBlock to prepare the user's system for execution of a given eStream application. It creates registry entries, copies files, and updates the file spoofing database. The user can then launch his application via a local shortcut or a shortcut on the eStream drive. Uninstallation involves undoing all changes made to the user's system by AIM during installation.

Data type definitions

This component uses the AppInstallBlock, but doesn't define it. This is defined in a low-level design document for the Builder component.

The AppInstallBlock is a binary data file with a versioned interface, basically consisting of:

- a header
- a list of files to install or send to the file spoofer
- a list of registry entries to install or remove
- a set of prefetch requests to communicate to the profile/prefetch component
- a set of initial profile data to communicate to the profile/prefetch component (post-version 1.0)
- a comment section
- an embedded DLL that can be loaded and executed for custom install needs
- a section containing a license agreement to be shown to the user

Many of the AIMsc functions take an AIBFileRef as an argument, which is an opaque pointer to the following structure:

```
typedef struct
{
    HANDLE          FileHandle;
    AIBFileHeader   FileHeader;
    AIBIndexEntry   *IndexEntries;
    LPCTSTR         AppName;
```

```
} AIBFileInfo, *pAIBFileInfo;
```

It is assumed that an external header file will be available that defines structures such as AIBFileHeader and AIBIndexEntry. For now, refer to the AppInstallBlock-LLD for how they might be defined.

Also, each application has a prefetch data file created for it an install time that is initialized with prefetch data from the AppInstallBlock. This data file is named and located as described in the Component Design section, and just consists of a non-padded list of the following structures:

```
typedef struct
{
   UINT32  FileNumber;
   UINT32  BlockNumber;

} PrefetchItem, *pPrefetchItem;
```

The following data types are used in the AIM and AIMsc interfaces:

```
typedef void *AIBFileRef;
```

Error codes that are assumed to be defined somewhere are:

```
SUCCESS (0)
ERROR_BUFFER_TOO_SMALL
```

Interface definitions

Application installation/uninstallation

There are only two functions exposed by AIM, one for application installation, and another for application uninstallation. Only the License Subscription Manager will be calling these functions.

UINT32 AIMInstallApplication(UINT8 AppId[16], LPCTSTR PathToAIB)

Parameters

AppId

[in] The application ID of the eStream application to install.

PathToAIB

[in] Pointer to a null-terminated string that specifies a path to an AppInstallBlock file to install.

Return Values

SUCCESS (0) if all the actions specified in the AppInstallBlock were performed successfully, an error code otherwise.

Comments

None.

UINT32 AIMUninstallApplication(UINT8 AppId[16])

Parameters

AppId

[in] The application ID of an existing eStream application to uninstall.

Return Values

If the specified application ID is not recognized, or the original AppInstallBlock is not found, an error code will be returned. Otherwise, AIM will make an attempt to undo all of the actions it performed while installing this application. It will return SUCCESS (0) if it undid enough of these actions so that any future installation of the same application will succeed.

Comments

None.

AIM Sub-Component Interface

Much of the functionality required by the AIM design will be useful to the Builder testing framework as well. This functionality will be treated as a sub-component within the AIM component, called AIMsc, and will export a well-defined interface. That interface is defined as follows.

UINT32 AIMscOpenAppInstallBlock(LPCTSTR PathToAIB, AIBFileRef *pAIBFile)

Parameters

PathToAIB

[in] Pointer to a null-terminated string that specifies a path to an AppInstallBlock file to open.

pAIBFile

[out] Returns a reference to an open AppInstallBlock file.

Return Values

SUCCESS (0) if the AppInstallBlock was opened successfully and validated, an error code otherwise.

Comments

The reference returned by this function can be used as a parameter to any of the other functions that take an AIBFileRef.

UINT32 AIMscCloseAppInstallBlock(AIBFileRef AIBFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

Return Values

SUCCESS (0) if the close succeeded, an error code otherwise.

Comments

None.

*void AIMscGetAIBVersion(AIBFileRef AIBFile, UINT32 *pAIBVersion)*

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersion

[out] Returns the value of the AibVersion field in the AppInstallBlock.

Return Values

SUCCESS (0) if the value was successfully retrieved, an error code otherwise.

Comments

None.

void AIMscGetAIBAppId(AIBFileRef AIBFile, UINT8 pAIBAppId[16])

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersion

[out] Returns the value of the AppId field in the AppInstallBlock.

Return Values

SUCCESS (0) if the value was successfully retrieved, an error code otherwise.

Comments

None.

*void AIMscGetAIBVersionNo(AIBFileRef AIBFile, UINT32 *pAIBVersionNo)*

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersionNo

[out] Returns the value of the VersionNo field in the AppInstallBlock.

Return Values

SUCCESS (0) if the value was successfully retrieved, an error code otherwise.

Comments

None.

void
AIMscGetAIBShouldReboot(
AIBFileRef AIBFile,
BOOLEAN *pAIBShouldReboot)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBShouldReboot

[out] Returns the value of the ShouldReboot flag in the AppInstallBlock.

Return Values

SUCCESS (0) if the value was successfully retrieved, an error code otherwise.

Comments

None.

UINT32
AIMscGetAIBAppName(
AIBFileRef AIBFile,
LPTSTR pAIBAppName,
UINT16 *pSizeAIBAppName)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBAppName

[out] The value of the ApplicationName field in the AppInstallBlock is copied into the memory pointed to by this address (it will be null terminated).

pSizeAIBAppName

[in, out] On input, should point to the size of the memory at *pAIBAppName*. On output, will point to the total bytes needed to hold the entire string if ERROR_BUFFER_TOO_SMALL is returned, otherwise is undefined.

Return Values

SUCCESS (0) if the value was successfully retrieved, ERROR_BUFFER_TOO_SMALL if the buffer is too small to hold the entire string, or another error code otherwise.

Comments

None.

UINT32
AIMscCheckAIBCompatibleOS(
AIBFileRef AIBFile,
BOOLEAN *pWasOSCompatible)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pWasOSCompatible

[out] Returns TRUE if the AppInstallBlock can be installed on the current OS, FALSE otherwise.

Return Values

SUCCESS (0) if the OS version was successfully retrieved and checked, an error code otherwise.

Comments

This function will check if the currently installed operating system and service is compatible with the specified AppInstallBlock (using the compatibility information contained in the AppInstallBlock). If not, it will display a detailed message to the user and return FALSE in *pWasOSCompatible*, otherwise it will do nothing and return TRUE in *pWasOSCompatible*.

```
UINT32
AIMscInstallAppFiles(
    AIBFileRef      AIBFile,
    HKEY            SpoofKey,
    HKEY            SpoofRefCountKey,
    LPCTSTR         InstallLogFile,
    BOOLEAN         *pIsRebootNeeded)
```

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

SpoofKey

[in] An open handle to the registry key where file-spoofing data is stored.

SpoofRefCountKey

[in] An open handle to the registry key where file-spoofing reference counts are stored.

InstallLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

pIsRebootNeeded

[out] Returns TRUE if a reboot is needed to complete the file copying, FALSE otherwise.

Return Values

SUCCESS (0) if all file install operations succeeded, an error code otherwise.

Comments

This function will perform the file copies and add the file spoofing entries specified in the File section of the AppInstallBlock. Changes will be appended to the install log file (it is created if it doesn't already exist).

For the sake of getting an error back to the user as soon as possible, this function will not undo file copies or spoof entry additions if it fails. **AIMscUninstallAppFiles** should be called to do so after the caller informs the user of the error.

UINT32
AIMscUninstallAppFiles(

AIBFileRef	***AIBFile,***
HKEY	***SpoofKey,***
HKEY	***SpoofRefCountKey,***
LPCTSTR	***InstallLogFile,***
BOOLEAN	****pIsRebootNeeded)***

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

SpoofKey

[in] An open handle to the registry key where file-spoofing data is stored.

SpoofRefCountKey

[in] An open handle to the registry key where file-spoofing reference counts are stored.

InstallLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

pIsRebootNeeded

[out] Returns TRUE if a reboot is needed to complete the file deletions, FALSE otherwise.

Return Values

SUCCESS (0) if enough of the file install operations were reversed so that re-installation will succeed and so that the system is in a consistent state. Otherwise, an error code is returned.

Comments

This function will reverse the file additions and remove the file spoof database entries specified in the install log file.

UINT32
AIMscInstallAppVariables(
AIBFileRef AIBFile,
LPCTSTR InstallLogFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

InstallLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

Return Values

SUCCESS (0) if all variable modifications succeeded, an error code otherwise.

Comments

This function will perform the add/remove variable (i.e. registry entry) changes specified in the Variable section of the AppInstallBlock. Changes will be appended to the install log file (it is created if it doesn't already exist).

For the sake of getting an error back to the user as soon as possible, this function will not undo registry modifications if it fails. **AIMscUninstallAppVariables** should be called to do so after informing the user of the error.

UINT32
AIMscUninstallAppVariables(
AIBFileRef AIBFile,
LPCTSTR InstallLogFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

InstallLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

Return Values

SUCCESS (0) if enough of the variable changes were reversed so that re-installation will succeed and so that the registry is in a consistent state. Otherwise, an error code is returned.

Comments

This function will reverse the add/remove variable (i.e. registry entry) changes specified in the install log file.

UINT32 AIMscInstallAppPrefetchFile(AIBFileRef AIBFile, LPCTSTR PrefetchFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

PrefetchFile

[in] A null-terminated string representing the path to the prefetch file to be created.

Return Values

SUCCESS (0) if prefetch block installation succeeded, an error code otherwise.

Comments

This function will install the prefetch information contained in the Prefetch section of the AppInstallBlock into *PrefetchFile*.

UINT32 AIMscUninstallAppPrefetchFile(AIBFileRef AIBFile, LPCTSTR PrefetchFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

PrefetchFile

[in] A null-terminated string representing the path to the prefetch file to be uninstalled.

Return Values

SUCCESS (0) if prefetch block uninstallation succeeded, an error code otherwise.

Comments

This function will remove the prefetch information stored at *PrefetchFile.*

UINT32 AIMscInstallAppProfileFile(AIBFileRef AIBFile, LPCTSTR ProfileFile)

UINT32 AIMscUninstallAppProfileFile(AIBFileRef AIBFile, LPCTSTR ProfileFile)

(NOT FUNCTIONAL IN ESTREAM 1.0)

UINT32 AIMscCallCustomInstall(AIBFileRef AIBFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

An error code if the custom code .dll could not be extracted, loaded and called. Otherwise, returns the value returned by the *Install()* function in the custom code .dll.

Comments

This function will extract and load the custom code .dll included in the Code section of the AppInstallBlock, and then call the exported .dll function named *Install()*.

UINT32 AIMscCallCustomUninstall(AIBFileRef AIBFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

An error code if the custom code .dll could not be extracted, loaded and called. Otherwise, returns the value returned by the *Uninstall()* function in the custom code .dll.

Comments

This function will extract and load the custom code .dll included in the Code section of the AppInstallBlock, and then call the exported .dll function named *Uninstall()*.

*UINT32 AIMscEnforceLicenseAgreement(AIBFileRef AIBFile, BOOLEAN *pBUserAgreed)*

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

pBUserAgreed

[out] Returns TRUE if the user agreed to the license terms, FALSE otherwise.

Return Values

SUCCESS (0) if the license agreement was successfully displayed, an error code otherwise.

Comments

This function will extract the license agreement text included in the LicenseAgreement section of the AppInstallBlock and display it to the user. The user will be given the option to agree or not agree to the license (probably via a pair of buttons in a dialog).

UINT32
AIMscDisplayComment(AIBFileRef AIBFile)

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

SUCCESS (0) if the comment was successfully displayed, an error code otherwise.

Comments

This function will display to the user the comment included in the Comment section of the AppInstallBlock.

Component design

AIMsc does not have hard-coded knowledge regarding any of the standard registry and file locations used by AIM, which is why the functions in its interface take as inputs specifiers for filenames and base registry locations. Conversely, AIM itself has no knowledge of the internal structure of the AppInstallBlock file, which is why it must call AIMsc functions to work with such files.

Expansion is perform on registry entries and file paths containing certain variables, when they are read from the AppInstallBlock. These variables are defined in the Builder-LLD and will be recognized and expanded by AIM. (This includes file-spoof entries.)

AIM stores its data in the expected places for an eStream client component. All of the data AIM stores is user-specific, so it makes no use of the global locations defined for eStreams.

Registry keys

AIM stores its registry keys and values under:

HKEY_CURRENT_USER\SOFTWARE\Omnishift\eStream\AIM

This key will have its permissions modified so that ordinary users cannot modify the key (but the eStream client service will be given privileges so that it can do so). Here are the subkeys AIM places under this key:

"SpoofEntries"

Spoof entries are placed here. All spoofing is done globally, so there is no need to place it under an eStream-app specific key. Each value under this key is a pair of pathnames as follows:

<old-pathname> (REG_SZ)
- <spoofed-pathname>

"SpoofEntriesRefCounts"

Reference counting for spoof entries is done here. If multiple eStream apps are installed that want to spoof the same file, the entries must be ref-counted so that uninstall does not break the other apps. Each value under this key is a pair like this:

<old-pathname> (REG_DWORD)
- <ref-count>

Every value under SpoofEntries has a value under SpoofEntriesRefCounts with the same value name.

"<AppId>"

Every installed eStream app has its own subkey whose name is a string representation of its AppId, like so: "{00000000-0000-0000-0000-00000000000}". The values stored under each such key are:

AppId (REG_BINARY)
- AppId in binary form (16 bytes)

AppName (REG_SZ)
- name of the application (same as in the AppInstallBlock)

AppInstallBlockPath (REG_SZ)

- path to the AppInstallBlock for the application

AppInstallState (REG_DWORD)

- a value of 0 means app is installed, 1 means install is in progress, 2 mean uninstall is in progress.

Files

AIM stores per-user files at the following path:

(Path to the user's home directory)\Application Data\Omnishift\eStream\AIM

For each installed application, a separate data folder is created. The name of the folder is the AppId of the application in GUID ASCII format, like so: "{00000000-0000-0000-0000-00000000000}". The files stored under each such folder are:

<GUID string>-AIB.dat	- the AppInstallBlock file for the application
<GUID string>-Prefetch.dat	- the prefetch data file for the application
InstallLog.txt	- a generated log of what to do during uninstall

The Prefetch data file is simply an array of PrefetchItem structures (as described in the Data Structures section).

The InstallLog.txt is a list of undoable actions taken during installation. This log will be used during uninstall to determine which files and entries are safe to remove. Each line in the file contains one change, and is of the form:

ADDED or OVERWROTE or SPOOFED FILE "<filename>" (fully qualified)

ADDED or OVERWROTE KEY "<keyname>" (fully qualified)

ADDED or OVERWROTE VALUE "<valuename>" (fully qualified)

AIMInstallApplication Prototype

Installing an eStream application consists of the following steps:

1. Preparing for the installation
2. Displaying a license agreement to the user and having him agree to it
3. Installing all required local files and spoof entries for this app
4. Setting/removing registry entries as required
5. Initializing the profile and prefetch data for this app
6. Performing any required custom installation tasks
7. Displaying the comment to the user if required
8. Completing the installation
9. Rebooting the computer if necessary

AIM's policy is that if it encounters any fatal error during the execution of AIMInstallApplication, it will attempt to undo everything it did before returning. AIM also gracefully handles aborted installs and uninstalls.

Step 1 – Preparing for the installation

First, AIM checks if the application is already installed by looking for an AppId registry key for the specified AppId. If found, then the AppInstallInProgress registry value is checked. If it exists and is 1, the user is asked if he wants to re-install, otherwise, he is asked to restart an aborted or damaged installation. If the user says no, **AIMInstallApplication** cleans up and exits with an error.

Next, a free disk space check is performed to ensure that enough disk space is available for the install. The available free space must be at least twice the size of the AppInstallBlock to proceed.

Next, an AppId folder is created for the app (described earlier), and the AppInstallBlock file is copied to this folder. AIM then opens the AppInstallBlock using **AIMscOpenAppInstallBlock**. Then the AppId registry key is created and the four defined values created and initialized. The AppInstallState value in particular is set to 1 to indicate an install is in progress. If any of these operations fail, **AIMInstallApplication** cleans up and exits with an error.

Step 2 – Displaying the license

AIMscEnforceLicenseAgreement is called to display the license text to the user and ask for his agreement. If the functions fails or if the user's response is returned as FALSE, **AIMInstallApplication** cleans up and exits with an error.

Step 3 – Installing local files

The install log file to be used for this application is created or open and truncated. **AIMscInstallAppFiles** is called to copy the install files to the computer and to create the spoof entries specified in the AppInstallBlock. Handles to the spoof subkey and the spoof refcount subkey are opened and passed to this function, as well as a path to the newly created install.log file. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

If it succeeds, a boolean is returned indicating whether a reboot needs to occur due to shared files being overwritten. This value is remembered for use in step 10.

Step 4 – Modifying the registry

AIMscInstallAppVariables is called to perform the registry modifications specified in the AppInstallBlock. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 5 – Initializing profile/prefetch data

AIMscInstallAppPrefetchFile is called to create and initialize the prefetch file for this application. The file has the structure specified in the Data Structures section of this document. This function takes a path to the prefetch file to be created. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 6 – Performing custom install tasks

AIMscCallCustomInstall is called to extract the custom code .dll contained in the AppInstallBlock and to call the *Install()* function it exports. If **AIMscCallCustomInstall** fails, **AIMInstallApplication** cleans up and exits with an error.

Step 7 – Displaying a comment

AIMscDisplayComment is called to display any comment to the user contained in the appropriate section of the AppInstallblock. If this function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 8 – Completing the installation

The AppInstallInProgress registry value is set to 0 to indicate the install is complete. **AIMscOpenAppInstallBlock** is called to close the AIBFileRef opened in step 1, and any handles to open registry keys are also closed.

Step 9 – Rebooting the computer (if necessary)

If **AIMscInstallAppFiles** in step 3 returned a value indicating a user reboot is necessary, or if **AIMscGetAIBShouldReboot** is called and returns a value of TRUE, the user is asked to reboot. Otherwise, no reboot is performed and the application is ready to be run. **AIMInstallApplication** exits returning SUCCESS (0).

AIMUninstallApplication Prototype

Uninstalling an eStream application consists of the following steps:

1. Preparing for the uninstallation
2. Undoing all modifications done to the registry during install
3. Undoing all file copies performed during install and removing spoof entries for this app
4. Deleting the profile/prefetch data for this application
5. Performing any required custom uninstallation tasks
6. Completing the uninstallation
7. Rebooting the computer if necessary

If the uninstallation fails for any reason, **AIMUninstallApplication** will tell the user that the uninstall has failed and that he should attempt to re-install the application before trying to uninstall again.

Step 1 – Preparing for the uninstallation

First, AIM checks if the application is already installed by looking for the AppId registry key corresponding to the specified AppId. If not found, then **AIMUninstallApplication** exits with an error.

Then, the AppInstallState value is set to 2 to indicate an uninstall is in progress. **AIMscOpenAppInstallBlock** is called to open the AppInstallBlock at the path specified by the AppInstallBlockPath key. If this fails, then **AIMUninstallApplication** exits with an error.

Step 2 – Undoing registry modifications

AIMscUninstallAppVariables is called to reverse the registry modifications specified in the AppInstallBlock. If the function fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

Step 3 – Undoing file copies and removing spoof entries

AIMscUninstallAppFiles is called to delete the files copied during install and to remove the spoof entries written then. Handles to the spoof subkey and spoof refcount subkey are passed to this function, and are where the spoof entries are removed from. If the function fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

Step 4 – Deleting profile/prefetch data

AIMscUninstallAppPrefetchFile will be called to remove the prefetch data stored for this application. Any failure is ignored.

Step 5 – Performing custom uninstall tasks

AIMscCallCustomUninstall is called to extract the custom code .dll contained in the AppInstallBlock and call the *Uninstall()* function it exports. If **AIMscCallCustomUninstall** fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

Step 6 – Completing the uninstallation

AIMscGetAIBShouldReboot is called and the return value saved. Then **AIMscOpenAppInstallBlock** is called to close the AIBFileRef opened in step 1, and the AppId folder and all its contents are deleted. The AppId registry key and all its subkeys are deleted also. Any handles to open registry keys are closed. Any failures here are ignored.

Step 7 – Rebooting the computer (if necessary)

If **AIMscUninstallAppFiles** in step 3 returned a value indicating a user reboot is necessary, or if **AIMscGetAIBShouldReboot** is called and returns a value of TRUE, the user is asked to reboot. Otherwise, the uninstallation is complete. **AIMUninstallApplication** exits returning SUCCESS (0).

AIMsc Function Prototypes

Prototypes for the AIMsc functions declared earlier are given in this section.

UINT32
AIMscOpenAppInstallBlock(LPCTSTR PathToAIB, AIBFileRef *pAIBFile)

First, the file at *PathToAIB* is opened. Then, the header is read in, header version and size is verified, and section sizes and offsets are verified. An opaque pointer to an AIB-FileInfo structure is returned in the pAIBFile parameter.

UINT32
AIMscCloseAppInstallBlock(AIBFileRef AIBFile)

The file handle at ((AIBFileInfo *) AIBFile)->FileHandle, and the AIBFile structure is freed.

void
AIMscGetAIBVersion(AIBFileRef AIBFile, UINT32 *pAIBVersion)

void
AIMscGetAIBAppId(AIBFileRef AIBFile, UINT8 pAIBAppId[16])

void
AIMscGetAIBVersionNo(AIBFileRef AIBFile, UINT32 *pAIBVersionNo)

```
void
AIMscGetAIBShouldReboot(
   AIBFileRef        AIBFile,
   BOOLEAN           *pAIBShouldReboot)
```

```
UINT32
AIMscGetAIBAppName(
   AIBFileRef AIBFile,
   LPTSTR    pAIBAppName,
   UINT16    *pSizeAIBAppName)
```

These four functions are trivial. They directly return the corresponding value of the variable in ((AIBFileInfo *) AIBFile)->AIBFileHeader. (See the interface declaration for **AIMscGetAIBAppName** for details on its calling logic.)

```
UINT32
AIMscCheckAIBCompatibleOS(
   AIBFileRef       AIBFile,
   BOOLEAN          *pWasOSCompatible)
```

This function will call an API such as GetVersionEx (for Windows) to determine the currently running operating system. The OS version is then converted to a bitmask (using constants defined in an external AppInstallBlock header file) and compared with the OS and Service Pack bitmaps in ((AIBFileInfo *) AIBFile)->AIBFileHeader. If the bits are present, *pWasOSCompatible* is set to TRUE, otherwise FALSE.

```
UINT32
AIMscInstallAppFiles(
   AIBFileRef       AIBFile,
   HKEY             SpoofKey,
   HKEY             SpoofRefCountKey,
   LPCTSTR          InstallLogFile,
   BOOLEAN          *pIsRebootNeeded)
```

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the File section. If not found, an error code is returned. Otherwise, the section is parsed.

The File section is organized as a series of trees, with directories as non-leaf nodes and plain files as leaf nodes. All nodes are stored contiguously according to the pre-order traversal of the trees.

Each directory node contains the name of a single directory and a number indicating the number of children this node has. Each file node contains the file version and file name, and a flag indicating whether the file is to be spoofed or not. If so, then the last entry in the node is the spoofed pathname, otherwise it is the actual contents of the file itself.

(The actual structure types defined for these nodes are assumed to be defined in a header file external to AIM. See the AppInstallBlock-LLD for reference.)

A directory stack algorithm will be used to parse the trees and reconstruct the directory paths. Due to the complexity of the task, several helper functions are used by the algorithm to partition this work.

For every file copied or spoof entry added by the algorithm, an entry is made to the file at *InstallLogFile*. For the sake of brevity, no mention is made of the logging in the pseudocode below.

The parsing algorithm is as follows (TOS refers to the node at the top of the stack):

```
empty out directory stack
while there are nodes to read in the File section
    read a node

    if the node is a directory
        HandleDirectoryNode(node, ...)
    else
        HandleFileNode(node, ...)

    while stack is non-empty and TOS node number of children is 0
        pop directory stack
        if stack is non-empty
            decrement number of children in TOS node
```

Here is HandleDirectoryNode(node):

```
if node directory name contains Builder/AIM defined variables
    replace variable substrings with local expansions

if directory stack is empty
    if node directory name is not fully qualified
        error
    push onto directory stack an entry with:
        -  node directory name
        -  node number of children
    else
        push onto directory stack an entry with:
        -  "TOS directory name" cat "directory name"
        -  node number of children
```

Here's HandleFileNode(node, ...):

```
if node filename contains Builder/AIM defined variables
    replace variable substrings with local expansions
```

```
if directory stack is empty
    if node filename is not fully qualified
        error
    call DoFileInstall(filename, node, …)
else
    if number of children in TOS node is <= 0
        error
    call DoFileInstall("TOS directory name" cat "filename", node, …)
    decrement the number of children in TOS node
```

Here's how DoFileInstall(filename, node, …) works:

```
if the file node is a spoof entry
    if filename already exists
        if existing version is earlier
            mark for spoofing
    else  // filename does not exist
        create zero-length file at filename
        mark for spoofing

else  // file will be copied not spoofed
    if filename already exists
        if this file is a .dll
            increment .dll shared ref count in registry
        if existing version cannot be read or existing version is earlier
            mark for copy
    else  // filename does not exist
        add line to FilesLogPath file containing filename
        mark for copy

if marked for spoofing
    create spoof entry under SpoofKey
    create or update spoof refcount under SpoofRefCountKey

if marked for copy
    attempt to copy node file to client computer
    if copy fails
```

```
        tell system to perform copy at reboot
```

The *pIsRebootNeeded* argument will be set to TRUE if any file copies were scheduled to happen at reboot, FALSE otherwise. Additionally, if any spoof entries were added, then an IOCTL will be sent to the spoof driver asking it to reload the spoof database.

The shared .dll reference count mentioned in the algorithm above is stored in a standard place in the Windows registry. AIM will create or increment this reference count for every non-spoofed .dll included in the AppInstallBlock (they can all be potentially shared since they will be placed outside of the eStream app directory). Each such .dll has an associated REG_DWORD value under the key at:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs

The value's name is the path to the .dll and the value's data is a integer that is the reference count for this .dll.

UINT32
AIMscUninstallAppFiles(

AIBFileRef	**AIBFile,**
HKEY	**SpoofKey,**
HKEY	**SpoofRefCountKey,**
LPCTSTR	**InstallLogFile,**
BOOLEAN	***pIsRebootNeeded)**

Currently, the *AIBFile* parameter is not even needed since all of the information needed for file uninstall is contained in the log file at *InstallLogFile*. However, it is included to enforce a design decision, which is that the user should have a valid reference to an AppInstallBlock before performing any install/uninstall related actions on an eStream app.

The algorithm for **AIMscUninstallAppFiles** is simple. It iterates over the change entries contained in the log file, and undoes file copies and spoof entry additions when it is safe to do so. Here is the algorithm:

```
while there are change entries in the log file
    read the next entry

    if the entry is of the form "ADDED <filename>"
        if filename is a .dll
            decrease refcount of .dll
            if refcount is 0
                mark for deletion
```

```
            else // file not a .dll
                mark for deletion

            if file is marked for deletion
                attempt to delete file
                if deletion fails
                    tell system to perform deletion at reboot

        else if the entry is of the form "OVERWROTE <filename>"
            if filename is a .dll
                decrease refcount of .dll

        else if the entry is of the form "SPOOFED <filename>"
            decrease refcount of spoof entry for this filename
            if refcount is 0
                delete the spoof entry
                if 0-byte placeholder at filename still exists
                    delete it
```

A failure to delete a file or to schedule its deletion will not cause **AIMscUninstallAppFiles** to fail.

UINT32
AIMscInstallAppVariables(
AIBFileRef AIBFile,
LPCTSTR InstallLogFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Variable section. If not found, an error code is returned. Otherwise, the section is parsed.

The Variable section is organized as a series of trees, similar to how the File section is organized. There are two types of nodes, key nodes and value nodes. Registry keys can contain other keys and registry values, while registry values are just name/data pairs that are stored in keys. A registry value name is always a string, but its data can be stored as any one of a number of types.

Non-leaf nodes must be key nodes, while leaf nodes can either by key or value nodes. All nodes are stored contiguously according to the pre-order traversal of the trees.

(The actual structure types defined for these nodes are assumed to be contained in a header file external to AIM. See the AppInstallBlock-LLD for reference.)

A keyhandle algorithm will be used to parse the trees and create the registry keys and values. Due to the complexity of the task, several helper functions are used by the algorithm to partition this work.

For every key or value added by the algorithm, an entry is made to the file at *InstallLogFile*. For the sake of brevity, no mention is made of the logging in the pseudocode below.

The parsing algorithm is as follows (TOS refers to the node at the top of the stack):

```
empty out the key handle stack
while there are nodes to read in the Variable section
    read a node

    if the node is a key
        HandleKeyNode(node, ...)
    else
        HandleValueNode(node, ...)

    while the stack is non-empty and the TOS number of children is 0
        if TOS key handle is open
            close it
        pop the directory stack
        if the stack is non-empty
            decrement the number of children in TOS
```

Here is HandleKeyNode(node):

```
if the key name contains a Builder/AIM defined variable
    replace the variable substring with its local expansion

if the keyhandle stack is empty
    if the key name is not fully qualified
        error
    create key under HKCR, HKLM, etc. and save key handle
else
    if the number of children in TOS is <= 0
        error
```

```
        create key under TOS key handle and save key handle

    push onto the keyname stack an entry with:
        -   open key handle
        -   this number of children
```

Here's HandleValueNode(node, ...):

```
    if the keyname stack is empty
        error
    else
        if the number of children in TOS is <= 0
            error
        call DoInstallValue(TOS key handle, node, ...)
        decrement the number of children in TOS
```

Here's how DoInstallValue(key handle, value node, ...) works:

```
    if the value name contains a Builder/AIM defined variable
        replace the variable substring with its local expansion
    call SetValueEx(key handle, "value name", value type, value data, ...)
```

UINT32
AIMscUninstallAppVariables(
AIBFileRef AIBFile,
LPCTSTR InstallLogFile)

Currently, the *AIBFile* parameter is not even needed since all of the information needed for file uninstall is contained in the log file at *InstallLogFile*. However, it is included to enforce a design decision, which is that the user should have a valid reference to an AppInstallBlock before performing any install/uninstall related actions on an eStream app.

The algorithm for **AIMscUninstallAppVariables** is simple. It iterates over the change entries contained in the log file, and undoes registry key and value additions when it is safe to do so. Here is the algorithm:

```
    while there are change entries in the log file
        read the next entry

        if the entry is of the form "ADDED <keyname>"
```

if key at keyname (fully qualified) still exists
    delete it and all subkeys and values

else if the entry is of the form "ADDED <valuename>"
    if value at valuename (fully qualified) still exists
        delete it

Keys and values that were overwritten are not deleted, which is why those log file entries are not considered by the algorithm above.

A failure to delete one or more registry entries will not cause **AIMscUninstallAppFiles** to fail.

UINT32
AIMscInstallAppPrefetchFile(AIBFileRef AIBFile, LPCTSTR PrefetchFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Prefetch section. If one is found, the prefetch data is read in and written out into the file at *PrefetchFile* as an array of PrefetchItem structures (this structure will change to match how the prefetch data items are represented in the AppInstallBlock-LLD). Any existing file at *PrefetchFile* is overwritten.

Next, the Prefetch component is called to set up an association between the new application and its prefetch file.

UINT32
AIMscUninstallAppPrefetchFile(AIBFileRef AIBFile, LPCTSTR PrefetchFile)

The file at *PrefetchFile* is deleted and the Prefetch component is called to remove the association between the app being uninstalled and the prefetch file.

UINT32
AIMscInstallAppProfileFile(AIBFileRef AIBFile, LPCTSTR ProfileFile)

UINT32
AIMscUninstallAppProfileFile(AIBFileRef AIBFile, LPCTSTR ProfileFile)

(NOT FUNCTIONAL IN ESTREAM 1.0)

UINT32
AIMscCallCustomInstall(AIBFileRef AIBFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Code section. If one is found, the section is read in and written out again as a .dll library.

This library is loaded and the *Install()* function export is called (and its return value returned).

UINT32
AIMscCallCustomUninstall(AIBFileRef AIBFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Code section. If one is found, the section is read in and written out again as a .dll library. This library is loaded and the *Uninstall()* function export is called (and its return value returned).

UINT32
AIMscEnforceLicenseAgreement(
AIBFileRef AIBFile,
BOOLEAN *pBUserAgreed)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the LicenseAgreement section. If one is found, the license text is read in and displayed to the user in a dialog. The user will be asked to either agree or disagree with the license, and *pBUserAgreed* will reflect his decision.

UINT32
AIMscDisplayComment(AIBFileRef AIBFile)

The index entry array at ((AIBFileInfo *) AIBFile)->IndexEntries is scanned to find the Comment section. If one is found, the comment text is read in and displayed to the user in a dialog.

Testing design

Unit testing plans

AIM will be tested by a program that generates AppInstallBlocks designed to stress the component. AIM will be asked to install the given AIB and if successful, the resulting state of the system will be compared to the expected state had all the files and variables been installed correctly. An uninstall will then be performed and the system state also checked.

The focus of the testing will obviously be on the File and Variable sections. The other sections such as Code and Comments will be stressed also, but their boundary conditions are much simpler.

AIM's ability to gracefully handle aborted installs and uninstalls will also be tested.

Stress testing plans

The program described above can be deliberately tuned to create AppInstallBlocks of unusual size and organization. For example, AppInstallBlocks with thousands of files and registry entries, or files and entries with unusually long names, etc.

Coverage testing plans

In addition to the stress testing, deliberately malformed AppInstallBlocks will be generated by the test program to hit as much error-handling code as possible. AIM's data files and registry entries can also be deliberately mangled to help achieve this effect.

Cross-component testing plans

As soon as they are available, Builder-generated AppInstallBlocks will be tested to verify that the AIM is compatible with the Builder's output. As soon as the LSM and a browser plugin are available, the communication path from browser to LSM to AIM will be tested. As soon as the file spoofer is available, compatibility with the file spoof entries that AIM makes will be tested.

Upgrading/Supportability/Deployment design

AIM will make use of the eStream logging facility to record information about errors and other unusual conditions that occur. The log file will be useful for diagnosing problems that occur during testing and in real world situations.

If the AIM component is upgraded, it must still be able to uninstall any eStream applications installed at the time of upgrade. This entails being able to interpret old AIM registry entries and data files, including the AppInstallBlock. This is more a concern for future designers of the AIM component, however.

Open Issues

- How will the various anti-piracy strategies being considered affect the design of AIM, if at all?

Client Senarios - Install, Upgrade and Configure

Version 1.0

[redacted]

Note that these sequences do not include the various places at which the installer should stop and ask for input from the user. It is intended to describe the installation and upgrade process from a technical standpoint. The UI must be described elsewhere.

Note that not all decision points are described. I ignore ones that don't have a material impact on the scenarios at hand. For example, when I say that the user downloads the eStream client installation program from the ASP's web site, this does not preclude the installer being delivered on other media, such as via ftp or physical media. It also includes the case of the system administrator acting on behalf of the user. When I say the client installer is unpacked to the user's disk, this covers any mechanism by which the user might run the installer, such as from a network share.

The packaging of the eStream client software installer is platform-dependent. On Windows, the installer will most likely be distributed as a .ZIP file or a self-extracting executable. On Unix platforms, the installer will probably be a .RPM or a gzipped tar archive.

Scenario 1- Install eStream client SW, no previous installation

This scenario is for the normal installation process. The machine does not have eStream installed on it. Client installation should only be performed once per machine (assuming that the client is not uninstalled.)

0. (Not shown on diagram.) ASP makes eStream client software available on its web site. The client SW install package may be generic, or it may be pre-customized for the ASP with settings for the ASP's servers.

1. The user downloads the eStream client install application from an ASP via a web browser.

2. The installer is copied to the user's hard drive and extracted. (Optionally, the installer can be accessed directly from a network share, as is the likely case in a corporate intranet.)

3. The installer is run the by the user (or by the system administrator on behalf of the user.) The installer queries the registry to determine if the eStream client has already been installed on this machine. Since this is a new installation, it will find that the registry keys for the eStream client are undefined. If there are registry settings for other installed software that may influence what the eStream installer will do, those are checked at this step as well. Note that the installer may choose different actions or to install different things depending on the version of the operating system that is present.

4. Just to be sure, the installer checks to see if any of the eStream client drivers are already installed on this system. (Note that we thus need some mechanism for determining which of our drivers is installed on a client system, and the versions of these drivers. Two possibilities are to look for the files on disk or to attempt to access the loaded drivers.) Since this is a new installation, none of the drivers will be on the system.

5. The installer copies the eStream drivers to the appropriate places on the user's system so that they can be loaded on the next system reboot. (If the drivers can be installed without rebooting the system, this is preferred.)

6. Client user-mode components are installed on the client system, in a user-specified location. The browser plugin (if we provide one) is installed for the user's default browser, or optionally, for any other supported browser.

7 System files (the registry and possibly other files) are modified so that the drivers will be loaded on the next system reboot. System files are modified so that the user-level components will be started on boot or logon, if that has been requested. Default configuration information (either specified by the user, or as customized by the ASP) will be written into the appropriate eStream configuration files, or into the registry. Uninstall information is written into the appropriate place.

8. Initial (empty) versions of the cache, application registry information, and application spoof information are created or installed. Alternatively, the client software could know how to create these files if they are not found when the software is started.

9. (Not shown on diagram.) The drivers are loaded into the running system, or the system is rebooted.

Needed APIs

None specifically needed.

Scenario 2 - Upgrade eStream client SW (trivial case - same or newer version installed)

This scenario covers the case when the user attempts an install or an upgrade of the eStream client components, when the version already installed is at least as new as the version they are attempting to install. This may occur if a user does not know that eStream is already installed on a machine, or if they download and run the installer to upgrade to the latest version, when they are already running the latest version. The installer or upgrader should determine that there is no point in doing an upgrade and gracefully exit. Note that Scenario 5 covers the case where the user elects to force a reinstall or a downgrade of the client software.

0. (Not shown.) ASP provides an eStream client software install or upgrade program via their web site.

1. User downloads the eStream client software from the ASP web server. The installation or upgrade program is for a version no newer than the eStream software already installed on the client system.

2. The install or upgrade SW is extracted to the client machine's hard disk.

3. The install or upgrade software checks the registry to see if there is an installed version of the client at least as new as the installer. There is, so the installer notifies the user that a newer version is already installed. The user elects not to reinstall the client software, and the installer exits.

Needed APIs

The installer or upgrader must determine the version of the client software currently installed. Most likely, this will be done through the registry, so no APIs are specifically needed.

Scenario 3 - Upgrade eStream client SW (easy case - no kernel components needed)

This scenario covers the case where the user installs an update to the eStream client software, but only the user-mode components are newer than those installed on the client system. In this case, it should be (theoretically) possible to replace only the user-mode components, and restart eStream without rebooting. This scenario is a special case of Scenario 4. We probably want to eliminate this scenario and always install fresh copies of all client system components (other than configuration related files).

0. (Not shown.) ASP provides an eStream client software install or upgrade program via their web site.

1. User downloads the eStream client software from the ASP web server. EStream is already installed on the client's machine.

2. The install or upgrade SW is extracted to the client machine's hard disk.

3. The install or upgrade software checks the registry, and finds that the installed client software is older than that provided by the installer. It also discovers that only user-mode components need to be replaced.

4. The installer checks to see if the user-mode client components are running, and if so, if any bits are currently being served through them. If so, it brings up a dialog box asking the user to shut down any applications that may currently be accessing the z: drive. The upgrade may be cancelled at this point. Kernel components are notified that user-mode components will be coming down for an upgrade. All non-kernel client components are shut down.

5. Client user-mode components are replaced, as necessary.

6. The registry is updated with information about the newly installed client components. Any necessary changes to the uninstall information are made.

7. Any persistent data (things in the cache, configuration files, etc.) that have changed format are converted to the new format, or discarded. (Alternatively, the client software could understand both the old and the new data format, and perform the conversion itself the next time it is run.)

8. (Optional). User-mode components are restarted. No reboot is necessary.

Needed APIs

The Cache Manager must support a stop client API:

bool StopClient()

StopClient would return TRUE if the client has been stopped, and FALSE otherwise (perhaps because the user is currently running an eStreamed app, and doesn't want to stop right now.)

Scenario 4 - Upgrade eStream client SW (most general case - kernel and user components)

This scenario covers the case when kernel-mode as well as user-mode components must be replaced. I didn't create a separate scenario for the case where kernel-mode components must be replaced, but user-mode components don't need to be, since that case is really no simpler than this one.

0. (Not shown.) ASP provides an eStream client software install or upgrade program via their web site.

1. User downloads the eStream client software from the ASP web server.

2. The install or upgrade SW is extracted to the client machine's hard disk.

3. The install or upgrade software checks the registry, and finds that the installed client software is older than that provided by the installer. It discovers that at least one kernel-mode component must be replaced.

4. The installer checks to see if the user-mode client components are running. If necessary, it brings up a dialog box asking the user to shut down any applications that may currently be accessing the z: drive. The upgrade may be cancelled at this point.

Kernel-mode components are notified that eStream is being taken down for an upgrade. All non-kernel client components are shut down, and drivers are unloaded, if possible.

5. Client user-mode components are replaced, as necessary. Kernel-mode components are also replaced. It may be necessary that new versions of the drivers are placed in a special location, to be installed on the next reboot.

6. The registry and other configuration files are updated with information about the newly installed client components. Any necessary changes to the uninstall information are made.

7. Any persistent data (things in the cache, configuration files, etc.) that have changed format are converted to the new format, or discarded. (Alternatively, the client software could understand both the old and the new data formats, and perform the conversion itself.)

8. (Not shown). Machine is rebooted, either immediately or some time later. On reboot, the new kernel-mode drivers are installed in the appropriate locations and loaded.

Needed APIs

See StopClient() above.

Scenario 5 - Forced Reinstall/Downgrade of client SW

Though we would normally prefer a user to uninstall eStream before installing an older version, forced reinstalls (and possibly downgrades) may be necessary for a variety of reasons. The most important situation in which we would want to support this is when the uninstaller fails for some reason, leaving the system in a partially-installed state. Note that users often need to reinstall software because configuration files or registry settings have become corrupted, leading to aberrant application behavior. Note that downgrades are not strictly safe, because newer versions may have made incompatible changes to persistent file formats. We should thus support a reinstall mode that replaces settings files with fresh, new ones.

0. (Not shown.) The ASP provides an eStream client installer on their web site.

1. User downloads the eStream client SW installer.

2. Installer is unpacked onto the client system's hard drive.

3. The installer queries the registry and determines that it is not newer than the version of the client software already installed. The user specifically requests a reinstall or downgrade.

4. The installer checks to see if the cache manager is running. If it is, it asks it to shut itself and all other user-mode components down. If necessary, the user will be prompted to shut down any applications currently accessing the z: drive.

5. If the cache manager was running, it notifies the kernel components that the user-mode components are shutting down for replacement.

6. (Not shown on diagram.) User components exit.

7. User-mode components are all reinstalled, possibly with an older version.

8. Kernel-mode components are all reinstalled, possibly with an older version.

9. Application setup information is overwritten with default values, including empty versions of persistent caches, app install information, and app spoof information. Optionally, the user could request that the configuration information and cache not be overwritten.

10. (Not shown.) The machine is rebooted immediately or later, and the new drivers are installed upon reboot.

Needed APIs

See StopClient() above.

EStream Client SW Configuration

We have not fully defined what aspects of the client software will be configurable. Certainly, cache size and location should be configurable.

0. (Not shown.) The user brings up the eStream configuration UI via some mechanism.

1. The UI determines the current configuration and settings by querying the cache manager. If necessary, the UI starts the cache manager.

2. The cache manager queries the running user-mode components and kernel-mode components to determine the current state of the system.

3. Some of the queries may end up referencing on-disk configuration information through the client file manager.

4. If the user changes a setting, the change is sent to the cache manager to take effect immediately.

5. Notices are sent to user and kernel mode components to take effect immediately.

6. Persistent changes are written to configuration files or the registry via the client file manager.

Needed APIs

We need APIs for the client UI to get and set program settings, and APIs to read and write these settings to persistent configuration files. The exact format of this APIs is a bit fuzzy now, since we don't know all of the things that we want to configure.

eStream Client Scenarios

[redacted]

Introduction

The following are scenarios or tasks we've identified that will take place on a client machine. Some of the following scenarios are explicitly requested by a user; others take place in the background without a user's knowledge.

Some working definitions and assumptions here:

- The eStream file system will be referred to as the Z: drive
- A "user" is registered with an ASP. An "account" is a "billable unit" with an ASP.
- For a given ASP, there's a many-to-many relationship between users and accounts (single user is a member of multiple accounts; an account is used by multiple users).
- A "user identifier" is cached info on a client machine for a user. It identifies
  - user, password
  - ASP
  - account server
  - DRM server

nstall/upgrade eStream client SW

he first time eStream is installed, this is an explicit action by the user or a sysadmin. For upgrades, this could be xplicit, or the eStream client could detect that it needs to upgrade itself (possibly by asking permission first). Some ubcategories:

- Install is requested, latest version of eStream already installed
- User doesn't have sufficient permission to install

onfiguration of eStream client SW

ither during installation, or subsequent to this, some client parameters need access.

- Size, location of on-disk cache
- Startup of eStream client: auto vs. manual

tart eStream client SW

his allows the client to query the account server for new subscriptions, to authenticate and start eStream'ed apps, to start y spoofers on the system, etc.

he client can start either explicitly by the user, or implicitly at boot or login.

Manage "user" and "account" data

Some subcategories:

Remove user data from an ASP

Add/remove account that this user is a member of

Subscribe/unsubscribe to an application

This can come from "within" eStream or not. I.e., the client SW may not be informed that a new subscription has been made (e.g., it's been done on the ASP web site); hence the subscribed app cannot be immediately installed.

Account/user queries

- billing info
- subscription info

"Install" account information on the client

This means to cache the "user identifier" on the client machine. This allows the eStream client do to asynchronous queries for new subscriptions, transparently start subscribed apps, etc.

nstalling account info is inherently an explicitly requested action. We identified two possible scenarios:

- using a dialog to identify and "log in" to an ASP account server
- export an existing "user identifier" to a file that could be transferred to another machine, and used by the eStream client to install the same identifier on this second machine (e.g., by double-clicking on it)

'Install" a subscribed application

his means: download all necessary bits to make a subscribed app "ready-to-run."

his can be explicit (e.g., when a user subscribes to a new app) or implicit (e.g., the client SW pings the account server to ook for a new subscription).

uery Z:\ (i.e., the root directory of the eStream FS)

Ve shouldn't require authentification if only the top-level components of the root directory are queried. Ideally, the client W simply identifies the contents with some representation of the actual directories -- e.g., with a special icon displayed Explorer.

uery Z:\.* (i.e., a file/directory under the root directory)

his includes actually running applications!

ccessing actual application directory contents:

- May need to authenticate before granting access
  - authentication may fail
- May not need to authenticate; e.g.,
  - already granted within time slice

- o already using files within the application hierarchy

Aim Explorer at a folder with a file that's associated with an eStream'ed app

Termination of an application

- Is this the last open file associated with an application hierarchy?
- Do we need to upload user data?
- Do we need to contact DRM server to give up authentication token?

Uninstalling components

Uninstalling a subscribed app

Unclear how clean we need to leave the system

Uninstalling eStream client components

Unclear how clean we need to leave the system

Failures/errors

App crashes

eStream crashes

Client machine crashes

Kill a zombie connection

Unexpected loss of connectivity

eStream™ Client FSD Design

version 0.2, 5 [redacted]

Curt Wohlgemuth
Omnishift Technologies, Inc.
Company confidential

Introduction

This is an initial requirements/design document for the eStream file system driver (EFSD), implementing the eStream file system (EFS), for a Windows 2000 client machine.

Very high-level view

On the client side of the eStream product, the user will have a networked drive visible and accessible; this drive will "contain" all applications available in the current user's session. There will be limitations on what will be visible to a user within this drive, and there are uncertainties as to how the entire client design will lay out. However, it's still useful to specify some known issues and suggestions for how the EFSD should be designed.

Here's the general overview of what will reside on the client and what the control flow will be.

- The eStream drive will be managed by the eStream FSD, which is a form of network redirector. It will interface with the I/O manager, Cache manager, and VM manager.
- Requests for file and directory contents will be passed from the EFSD to what I'm calling the "eStream Client Manager," or ECM, which may reside in either user or kernel space.
- The ECM will handle, at least:
    - passing necessary read/write data and metadata requests to the server
    - caching all pertinent data and metadata
    - performing any profiling and prefetch work
    - helping the eStream FSD keep coherence with the server
- The network interface between the ECM and the server is the subject of other design documents

Although in practice it may be that much of the eStream drive will be read-only from a user's perspective, the EFSD must be designed for full write capabilities. Policies for read/write access will be the responsibility of the ECM and the server.

Design overview

Interfacing with the client OS

The W2000 EFSD will handle all appropriate requests from the various Windows Executive components. Here is a list of requests it needs to handle:

- Create IRPs, for both new and existing files
- Cleanup, Close IRPs
- Read and Write IRPs:
    - synchronous and asynchronous
    - cached and non-cached
    - paging and non-paging
- Fast I/O reads and writes (with buffers or MDLs)

- File information (get and set) IRPs
- Directory query IRPs
- Volume information (get and set) IRPs
- File system information (get and set) IRPs
- Various Device control IRPs (as needed by our implementation)
- Flush buffer IRPs
- System shutdown IRPs
- Various Fast I/O queries

In particular, I expect we do **not** need to handle Directory Notification IRPs, though this is potentially an open issue.

I'm also proposing we not support hard links on a Windows client (these are supported natively on NTFS on W2000 only).

The question of Byte-lock IRPs is an open one. These would probably be expensive to properly support, but it's unclear this time if not supporting them will be a showstopper for us or not.

Interfacing with the ECM

How the eStream FSD will interface with the ECM will depend on whether the ECM is a user or kernel space service. Nevertheless, the data that needs to be exchanged between the two components can be described today.

The EFSD and the ECM need to communicate all the basic information above, including:

- Create requests
- Open requests
- Read/write requests
- Directory content requests
- Rename requests
- Delete requests
- File/directory metadata requests
- Buffer flush requests
- Volume allocation information requests (possibly)

n addition, the ECM must be able to inform the EFSD that a file or directory it has open is either:

- not being modified by another process on any system; or
- potentially being modified by another process on another system

'his is necessary for the EFSD to allow caching of the file on the client machine.

Vhat's cached where?

)ue to the requirements of the NT executive components, the EFSD needs to cache various items of file metadata in its ·wn data structures. As long as it's tracking some attributes, it makes sense to keep track of all that it easily can. In ·articular, the eStream FSD will keep track of:

- Time stamps: creation, last write, last access, last change other than write
- File sizes: allocated, actual (aka EOF)
- File attributes: e.g., normal, directory, read-only, hidden

'he EFSD will not cache directory contents information; this will be the task of the ECM. Given how frequent these ›quests come through to the file system on Windows, we will definitely need to cache this on the client side.

eStream FSD design points

In no particular order, here are some items that must be implemented in the eStream FSD.

1. There are no volumes, and no VPB for a network redirector; I've verified this with the LanManager redirector.
    a. We do need to understand how we can have the EFSD handle multiple mounted drives if needed, and how to distinguish them by name in the Create IRP.

    b. We don't have to support any operations on a volume in EFS.

2. We should disallow the creation of paging files in EFS. There is a bit available for a Create IRP that specifies this, and we can complete the IRP with an unimplemented return code.
3. All file synchronization will be on an FCB basis, using the standard Resource and PagingIoResource ERESOURC objects used by the rest of the Windows Executive.
    a. User requests will be synchronized by acquiring the main Resource -- shared for reads, exclusive for writes, other changes, deletion, etc.
    b. Paging I/O requests will be synchronized by acquiring the PagingIoResource -- shared for reads, exclusive fc writes. Also exclusive access will be needed to set file sizes.
4. Most disk file systems have a resource associated with a VCB, which is acquired exclusively for creation/deletion etc. We will have a global EFS resource for this, since there are no VCBs.
5. Asynchronous requests will be handled by posting the IRP to the CriticalWorkQueue, and marking the IRP as pending.
    a. A common worker routine will be used for all async posts, which will dispatch the IRP to the appropriate rea IRP routine when it's invoked.
    b. An async request will be defined as one that IoIsOperationSynchronous() returns false, and the EFSD is the top-level component (see below)
6. The EFSD will track the top-level IRP for the thread whose context it is running in. In particular,
    a. No async processing request will be honored unless the EFSD is the top-level component
    b. No cache manager requests will be made unless the EFSD is the top-level component
    c. The top-level component is expected to acquire the appropriate resources, and the EFSD must not try to acquire them as well if it is not top-level
    d. EOF file size will not be extended or changed by paging I/O
7. EFS will not support holes in files, and hence the ValidDataLength FCB field will be set to disable this.
8. Most fast I/O routines will be supported in EFS. We should be able to use the FSRTL supplied routines for fast reads and writes.
9. All cache manager resource acquire/release callbacks will be supported. All will point to common routines that simply acquire or release the main Resource for the FCB. The Context pointer passed into all of them will be the FCB for the stream.
10. Synchronous read/write requests will update the CurrentByteOffset in the File object
11. Each Create will result in a unique CCB data structure; this will be small, and only hold those few fields needed:
    a. For the Directory Control IRP, a CCB needs to hold the current entry index and the pattern originally used -- for subsequent queries
    b. A field for various flags
12. A single FCB will represent all current open stream instances of a file. When a new file is opened, the EFSD will search the current open FCBs to find one matching this file/directory name.
    a. For now, this will be a simple linked list with a linear search. We can improve this as needed
    b. The EFS global resource must be acquired exclusively:
        i. before the global FCB "list" is searched
        ii. before a new FCB is added to the list
        iii. before an FCB is deleted from the list
13. EFS will not support open by file ID; hence the FileInternalInformation class for a File Information IRP will not be supported.
14. Actual I/O will be directed to standard routines in a separate file, so they can be isolated and updated easily as our method of transferring data changes.

15. Here's how to do file/directory renames:
    a. The I/O manager will send to the EFSD this sequence:
        i. Create for source
        ii. Create for target, with the SL_OPEN_TARGET_DIRECTORY flag set
        iii. Set Information with a Rename request for the source, sending the target directory FileObject handle, and the target filename in the file info record.
    b. EFSD needs to do this:
        i. When it receives the Create for the target and the target *directory* exists, return STATUS_SUCCESS, and change the name in the FileObject to the basename of the target (the full pathname of the target is sent in), and set the Status.Information to FILE EXISTS or FILE DOES NOT EXIST, as appropriate. If the target directory doesn't even exist, return PATH NOT FOUND.
        ii. When it receives the Set Info request, if all the flags check out (e.g., if the file exists, ReplaceExisting must be TRUE), send a Rename request to the ECM.
16. Reads and writes to only regular files will be supported, not to directories.
17. Any code that touches user buffers or can call routines that may throw exceptions must be guarded by a try/except block.
18. Some tips on memory allocation (from osrdocs/defensive-driv.html)
    a. Use our own memory allocation/deallocation routines, instead of ExAllocatePool() et al. directly
    b. These routines can do various checks for trashing memory:
        i. fill allocated memory with a pre-defined bit pattern, instead of zeroes; fill deallocated memory with a different pattern.
        ii. allocate a header/trailer with standard information, like where allocated, from what pool, etc.
        iii. change the bit pattern in the header/trailer on deallocation, and look for freeing memory twice

Design outline

The EFSD will look a lot like the sample FSD from Rajeev Nagar's NT FS book, which looks a whole lot like the FASTFAT FSD source from the IFS kit.

Most IRPs will have essentially two routines each to handle them: a dispatch routine which is sent the IRP directly, and a real routine that does the actual processing. The dispatch routine is essentially a stub that calls the real one. The real one is used to handle async processing of the IRP by a worker thread.

DriverEntry

This does a whole slew of initialization, including the dispatch table, fast I/O table, the cache callbacks, the FCB list and its synchronization object, creates the FS device object, and sets up the interface with the ECM.

Create

There is one Create routine; there will be no async processing of Create requests. Ultimately, its job is to send a create or open request to the ECM, and return SUCCESS or not to its caller. It also does at least the following:

- The global FCB data structure synchronization object must be acquired exclusively
- Paging files are not allowed
- Write-through requests result in no caching
- Related file objects must be for a directory
- An absolute pathname will be generated if a related file object is specified
- It will request to open/create the file from the ECM, and if successful, get the attributes it needs for the FCB
- The FCB must be searched for by name in the global data structure; if found, this is used, else a new one created.
- A new CCB must be created for this file
- If open existing entry is specified, and not found, the correct error must be returned (e.g., FILE NOT FOUND vs. PATH NOT FOUND)
- If OPEN_TARGET_DIRECTORY is specified, EFSD must request an open for the target directory from ECM, and

confirm its existence. The FileObject's name (from the IRP) must also be munged as specified above.
- The Share Access value sent in must be checked against the existing one for the FCB if this is not the first stream associated with this FCB.
- If any error is found, all data structures allocated must be released.
- The IsFastIoPossible member is set to FastIoIsPossible.

Cleanup

There may be async posting of Cleanup requests. Some points:

- The global resource and the FCB Resource must be acquired exclusively, posting if necessary
- This represents the end of processing for this stream, but not a close of the file.
- If caching is on, the cache must be flushed, and the pages purged.
- The count of open handles in the FCB must be decremented
- Any time stamps must be updated if accesses were done using fast I/O.
- The FO_CLEANUP_COMPLETE flag in the FileObject must be set
- IoRemoveShareAccess() must be called

Close

There may be async posting of Close requests.

- The global resource must be acquired exclusively
- The CCB must be deallocated
- The file's time stamps must be updated if the CCB bits indicate access/modification
- If this is the last reference to the FCB for the file, the FCB is deallocated
    - any updated file attributes must be pushed back to the ECM in this case
- If the file is marked to be deleted on close, the ECM must be asked to delete the file

Read

Reads will definitely be open to async posting. Some points:

- Non-buffered reads need to go straight to the ECM to request data; buffered reads must request data from the cache
- For now, we'll use the standard copy interface for cache reads
    - CcReadMdl() for MDL reads
    - CcCopyRead() otherwise
        - **Note:** the buffer to use might be for an allocated MDL, or it might be the UserBuffer!

- If this is non-paging, non-buffered, and the file stream is also open for buffering, need to:
    - grab the PagingIoResource exclusive
    - purge the cache for the range of the FCB
    - release the PagingIoResource
- Synchronization: grab the main Resource shared for non-paging reads; else the PagingIoResource shared.
- Zero length reads always return success
- Reads starting beyond EOF return EOF
- Requested lengths will be truncated to size if the request goes beyond EOF
- The cache map must be initialized if this is the first buffered read for the FCB
- For synchronous, non-paging reads, update the CurrentByteOffset in the CCB
- For non-paging reads, a bit in the CCB must be set to indicate that the file was accessed

Write

Writes will definitely be open to async posting. Some points:

- Non-buffered and buffered writes are as above for reads.
- If this is a buffered write, need to call CcCanIWrite(); if false, call CcDeferWrite() and post for async processing.
- A non-paging, non-buffered write for which the file stream is also open for buffering means:
  - grab the PagingIoResource exclusive
  - purge the cache for the range of the FCB
  - flush the cache as well (**Note: additional reqt!**)
  - release the PagingIoResource
- Writes of length zero immediately succeed
- A byte length of Low == FILE_WRITE_TO_END_OF_FILE, High == 0xffffffff signifies to start at EOF
- For paging writes:
  - No write requests beyond EOF will be honored
  - If the starting offset is EOF or beyond, just return success
  - If ending offset is beyond EOF, truncate write length to EOF
- For buffered writes:
  - If the write will extend the file size, inform the cache manager about the size change
  - If this is an MDL write, use CcPrepareMdlWrite(), else use CcCopyWrite().
  - Again, as for reads, even for a non-MDL write, there may be an MDL allocated that we should use.
- For synchronous, non-paging writes, update the CurrentByteOffset in the CCB
- For non-paging writes, set a bit in the CCB specifying that the file has been modified.

Fast I/O Read

Initially at least, we'll just set the fast I/O read routine to FsRtlCopyRead().

Fast I/O Write

Initially at least, we'll just set the fast I/O write routine to FsRtlCopyWrite().

Fast I/O Query Basic Info

This will just fill in the basic info buffer with the data in the FCB.

Fast I/O Query Standard Info

This will just fill in the standard info buffer with the data in the FCB.

Fast I/O Query Open

This will just fill in the network open info buffer with the data in the FCB, if the file exists. Some empirical observations 've made using NTFS:

- Regardless of whether the file exists or not, this will return TRUE (all fast I/O routines are boolean)
- If the file does not exist, it will set the EOF size in the buffer to 0. The AllocationSize must be non-zero. All other fields seem to be don't cares.
- If the file exists but is zero length, then both EOF and AllocationSize will be 0.
- The IRP sent to this routine is for an IRP_MJ_CREATE; we can use more than just the name to identify the file, bu also the security characteristics or whatever else is sent in the IRP.

File Query Info

Standard queries will be supported; these however will not:

- FileInternalInformation -- no OPEN_BY_FILE_ID
- FileEaInformation -- no EA data
- FileCompressionInformation -- no on-disk compression
- FileStreamInformation -- no multiple streams

File Set Info

These actions will be supported:

- EndOfFile size changes
- Time stamp changes
- File position changes
- File disposition changes -- delete pending
- File rename requests

In other words, all standard file set requests will be honored except AllocationSize changes.

Directory Query

This is an ugly NT interface, and unfortunately one that we need to expose across the ECM interface as well. Some points:

- These requests come in from the I/O Manager in a context-sensitive sequence. I.e., a request will come for the initial N directory entries; the next request will be for the next M entries; etc. Kind of like strtok().
- Thus, state must be maintained from request to request. This state will be kept in the CCB for a file stream, and consists of:
    - Pattern sent in on first request
    - Index of n'th entry to start retrieving with
- My experience is that the INDEX_SPECIFIED flag is **never** set in a directory control query, even on queries subsequent to the initial one.
- Here's the algorithm as best I can grok it from the fastfat sources:
    - If the CCB pattern field is empty, and the CCB flag specifying "match all" isn't set, this is the initial query
    - For the initial query, the main FCB Resource must be acquired exclusive, else we must acquire it shared. **WHY? If we're only modifying the CCB, why lock the FCB?**
    - The user has specified an index if the SL_INDEX_SPECIFIED IRP flag is set; we're ignoring all but the first match if SL_RETURN_SINGLE_ENTRY is set.
    - If this is the initial query, parse the pattern. If "*", set the "match all" flag in the CCB. In any case, save the pattern in the CCB.
    - Start with the current index in the CCB; for the initial query, this is 0; for subsequent queries, this is the inde› saved in the CCB. Both of these are overruled by the SL_INDEX_SPECIFIED flag and value.
    - Both are also overruled if the SL_RESTART_SCAN flag is set; index goes back to 0.
    - After filling the values for an entry into the supplied buffer, update the CCB index field.
    - Fill the input buffer with as many entries as possible. If we run out of space, stop, and return STATUS_SUCCESS.
    - If there isn't space for the very first entry for this query (base length of record + filename), return STATUS_BUFFER_OVERFLOW.
    - Total number of bytes written to user buffer is returned in Status.Information.
    - If there are no entries in the initial query, return STATUS_NO_SUCH_FILE.
    - If there are no entries in subsequent queries, return STATUS_NO_MORE_FILES.
    - The FileIndex field of the records returned in the buffer must be fixed up to be the **byte** offset from **this** record to the **next** record in the buffer. The FileIndex of the **last** record returned in the buffer should be 0; however, it appears from the fastfat sources that this isn't the case; they update the FileIndex value to the next entry offset, even if this can't fit in the buffer! This actually makes it a lot easier.
- In order to satisfy this EFS interface, we need a relatively similar interface between the EFSD and the ECM.

- I'm proposing that only a **single** directory entry at a time be transferred between ECM and EFS.
- We can further simplify things by essentially delivering only a FileBothDirectoryInformation buffer from ECM to EFS. This is a superset of anything the I/O Manager will request of EFS.
- *Someone* has to do pattern matching: either EFS, ECM, or the eStream server. It shouldn't be the server. I'r proposing it be ECM.

- Hence, the interface from EFS to ECM for directory queries could be:
  - Input:
    - directory handle
    - pattern
    - starting index
    - buffer
    - buffer length
  - Output:
    - (filled up buffer)
    - number of bytes filled in to buffer
    - if # bytes returned is 0, return code indicates either:
      - no entries available
      - an entry is available, but the buffer isn't large enough

- This should evenly divide the work on the client machine between the EFS and the ECM, and allow the ECM code to be more portable than if all the complexity were pushed onto it.

File System Query Info

Empirically, I've noticed that the LanMan redirector returns failure for most of these requests. So, except for any user-defined FSCTL requests we want to define, I'm going to fail all of these until it turns out we need to do otherwise.

File System Set Info

Ditto for this IRP type too.

Volume Query Info

We at least need to minimally implement these requests:

- FileFsAttributeInformation
- FileFsVolumeInformation
- FileFsDeviceInformation

Volume Set Info

We will fail all requests of this type.

Flush Buffers

A buffer flush request for a file stream will mean the following:

- If the file stream isn't buffered, return immediately
- The FCB main Resource is acquired exclusive
- The Cache Manager is told to flush the buffer for the byte range of the file
- The resource is released

A buffer flush for a directory is a successful NOP.

System Shutdown

IOCTLs

eStream Client Networking Low Level Design

Dan Arai
Version1.6
[redacted]

Functionality

The Client Network Interface (CNI) provides the interfaces for sending messages to servers and provides threads for receiving responses and dispatching them appropriately. It uses the eStream Messaging Service (EMS) APIs to send and receive various messages to and from the application servers and SLiM servers.

The number of threads in the CNI will depend on the functionality available from the EMS. In particular, more threads are necessary if the EMS provides asynchronous messaging capability (and the CNI uses this interface). The interfaces presented by the CNI are identical for both cases, but the internal organization of the component is not.

The prefetcher will make calls to client networking interfaces (indirectly through ECM-ReservePage) to send requests for pages. Similarly, the LSM will make calls to acquire access tokens and subscription information.

The networking component is responsible for examining the stream of requests to it and deciding when to coalesce multiple page requests into a single request to the server.

The EMS does not provide reliability in the event of server failure. The CNI is responsible for handling server failover and reissuing failed requests on different servers. The CNI abstracts the servers from other parts of the system. Clients of the CNI don't need to specify a particular server to make a request.

Since the client networking component is where timeouts and retries occur, it is the component that controls the policies for how long we wait for a connection to time out and how many times we retry a request before giving up. These parameters will be tunable. Any other parameters of the CNI that make sense to tune will be tunable.

The CNI is also the component responsible for implementing the server selection policy.

Data type definitions

The CNI uses the request structure defined by the ECM.

The CNI maintains an internal queue of messages that must be sent to servers. This queue is not exposed outside of the CNI. Like the ECM request queue, this queue will be maintained as a circular, doubly-linked list.

```
typedef struct _NWRequest
{
     NWRequestType type;
     union {} parameters; /* params, depends on type */
     struct _NWRequest *next;
     struct _NWRequest *prev;
} NWRequest;

typedef enum
{
     CNI_PAGE_READ,
     CNI_ACQUIRE_ACCESS_TOKEN,
     CNI_GET_LATEST_APP_INFO,
     CNI_RENEW_ACCESS_TOKEN,
     CNI_RELEASE_ACCESS_TOKEN,
     CNI_REFRESH_APP_SERVER_SET,
     CNI_GET_SUBSCRIPTION_LIST
} NWRequestType;
```

The CNI provides an enumeration of the parameters that can be tuned. This enumeration is expected to grow as the number of tunable parameters grows.

```
typedef enum
{
     CNI_NUM_RETRIES,
     CNI_TIMEOUT,
     CNI_PROXY_ADDRESS,
     CNI_EFFECTIVE_BANDWIDTH
} NWTunableParameter;
```

Related Components

The prefetcher and LSM call on the CNI to send requests to the app and SLiM servers. The CNI makes calls to the ECM and LSM to inform them of responses that have come back from the server. The CNI will also make calls to EFSD interface functions when pages come back that satisfy EFSD requests.

Interface definitions

CNIGetPage

eStreamStatus CNIGetPage(
  IN ApplicationID *app*,
  IN EStreamPageNumber *page*
);

CNIGetPage is the interface used by the ECM function **ECMReservePage** to request that a page be sent by the server. (**ECMReservePage** is called indirectly by the pre-

fetcher.) Note that no distinction is made between prefetches and demand fetches. To prevent race conditions or deadlock, the requested pages must already be marked as "in flight" in the index, and any requests for these pages from the EFSD must already be on the "in flight" queue before calling this interface.

The CNI is responsible for selecting a server to direct this request to, and resending in the event of network or server failure. It will coalesce requests for multiple pages from the same application into a single request to the server.

CNIGetSubscriptionList

eStreamStatus CNIGetSubscriptionList(
IN string *Username,*
IN string *Password*
);

CNIGetSubscriptionList enqueues a request to acquire a subscription list from a SLiM server. When the subscription list is returned by the server, the client response thread will notify the LSM of the returned data via a callback defined in the LSM document.

CNIGetLatestApplicationInfo

eStreamStatus CNIGetLatestApplicationInfo(
IN uint128 *SubscriptionID*
);

CNIGetLatestApplication enqueues a request to get the latest application information for a particular app. When the server returns the result, the CNI will notify the LSM of the returned data via a callback defined in the LSM document.

CNIAcquireAccessToken

eStreamStatus CNIAcquireAccessToken(
IN uint128 *SubscriptionID,*
IN string *Username,*
IN string *Password*
);

CNIAcquireAccessToken will cause the CNI to contact a SLiM server to retrieve an access token. The CNI is responsible for issuing retries if no response is received for a request. The CNI will call the appropriate LSM callback function when the data come back.

CNIRenewAccessToken

eStreamStatus CNIRenewAccessToken(
IN AccessToken *Token,*
IN string *Username,*
IN string *Password*
);

CNIRenewAccessToken will enqueue a request for access token renewal. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

CNIReleaseAccessToken

eStreamStatus CNIReleaseAccessToken(
 IN AccessToken *Token,*
 IN string *Username,*
 IN string *Password*
);

CNIReleaseAccessToken will enqueue a request for releasing an access token. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

CNIRefreshAppServerSet

eStreamStatus CNIRefreshAppServer(
 IN AccessToken *Token,*
 IN uint32 *BadQOS,*
 IN uint32 *NoService*
);

CNIRefreshAppServerSet will enqueue a request for refreshing the app server set. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

The client networking component will also have routines for getting and setting tunable parameters.

CNISetParameter

eStreamStatus CNISetParameter(
 IN NWTunableParameter *type,*
 IN void **value*
);

CNISetParameter sets a parameter. The actual type of *value* is determined by *type.*

CNIGetParameter

eStreamStatus CNIGetParameter(
 IN NWTunableParameter *type,*
 OUT void **value*
);

CNIGetParameter queries the current value of a parameter. The actual type of *value* is determined by *type.*

Component design

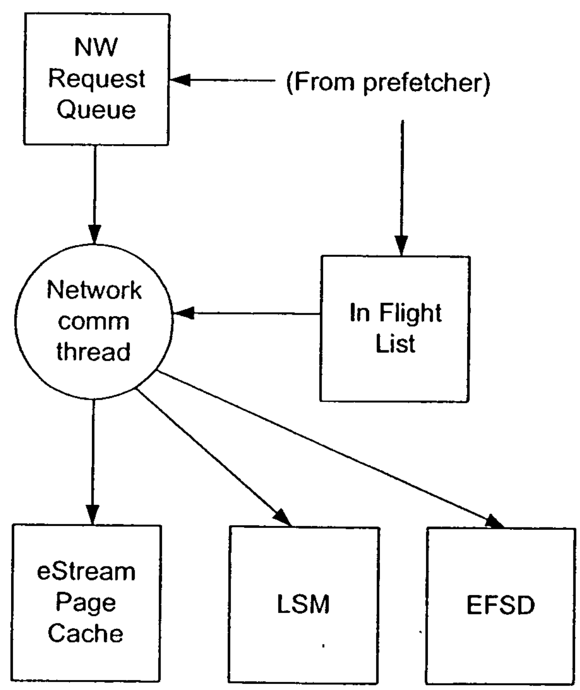
The internal organization of the client networking depends on the mechanisms available from EMS. Internally, the CNI interface functions put requests on a queue, and one or more threads services these requests by using the EMS to send messages to servers.

Synchronous Server Calls

If EMS only provides a synchronous messaging service, a single thread will be used to perform all necessary actions. The CNI interfaces will put appropriate requests on the network request queue. They will also wake up the network communication thread, if necessary.

The network communication thread's job is relatively simple. When it wakes up, it performs the following tasks:
- choose a set of requests to be coalesced and remove these from the request queue
- retrieve a server set via LSMGetAppServerSet or LSMGetSLiMServerSet, and choose a particular server for this request
- make a synchronous EMS call to send the request
- dispatch the response to the appropriate LSM or ECM callback

If the synchronous messaging mechanism becomes a performance bottleneck, we can have multiple network communication threads to increase concurrency.



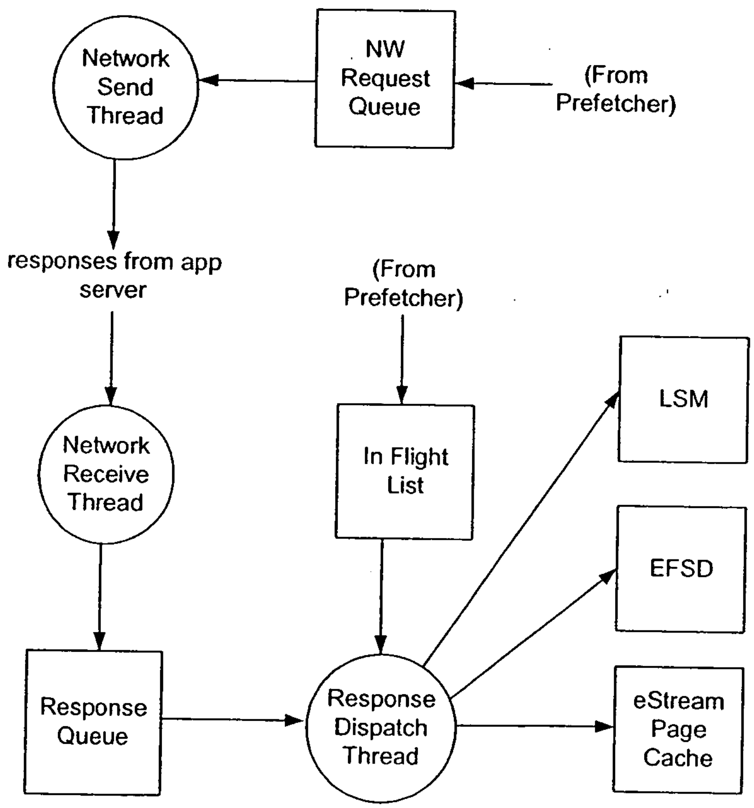
Asynchronous Server Calls

The asynchronous case is a little bit more complex. Because of the proposed asynchronous call architecture, the client NW requires three threads. The CNI interfaces work just as they do in the synchronous case. They put requests on the network request queue, and wake up the network send thread. However, the actions performed by the CNI's worker threads differ in the asynchronous model.

The network send thread is periodically awoken, and it coalesces requests off the NW request queue and sends them to the server. Unlike in the synchronous model, this thread does not synchronously wait for the request to come back from the server. Instead, it simply sends requests until the queue is empty, then goes back to sleep.

The network receive thread waits for responses to come back from any server. Because of the EMS's asynchronous call implementation details, this thread posts returned data to a queue of responses to be handled by another thread. The network receive thread is also responsible for handling timeouts and reissuing those network requests on different servers.

Finally, the response dispatch thread pulls responses off the response queue, and handles the work of dispatching them appropriately.



Handling Network Failure

When the client networking component is notified of a message failure by the EMS, the client worker thread will attempt to reissue the request on a different server.

Coalescing Multiple Requests

The CNI will coalesce multiple page requests that come from the LSM into a single request to an application server. Multiple pages requests for the same application may be coalesced. No other types of requests may be coalesced, including page requests for dif-

ferent applcations. The CNI will not produce requests larger than the maximum allowed by the application server.

Handling Persistent Failures
There will be some persistent failures that will result in the network being unable to fulfill page requests in a timely fashion. This may be due to network or server failure. (These may be indistinguishable from the CNI's point of view.) When the CNI has failed to satisfy a request for a certain amount of time, it will need to ask the user if he wants it to continue retrying, or if it should let the application terminate. It will do this via the **CUIAskUserYesNo()** interface. The client software control panel should include an option to always wait until the server is available, and never ask the user if he wants the application terminated.

Testing design

Unit testing plans

The testing harness for the networking component will be a set of dummy EMS drivers and a dummy NW client. The dummy EMS driver will be capable of performing a variety of actions, including returning appropriate responses, returning inappropriate responses, and timing out without any response. The dummy NW client will have knowledge about the expected EMS behavior, and will verify that the data it gets back from the network component are as expected.

Stress testing plans

Failure testing plans

The client NW is the sole component responsible for implementing server failover. In order to test this code, it is necessary to implement a server with predefined bad behavior. The server failure modes that must be tested include
- server that accepts a connection on a socket but doesn't respond to any requests
- server that closes the socket before sending a response
- server that closes the socket in the middle of a response
- server that sends a partial response and then just stops
- server that satisfies n requests then closes the socket or refuses to service more

It is important that we cover scenarios that look like network failures and ones that look like server failures. (Are there other failure modes that are interesting?)

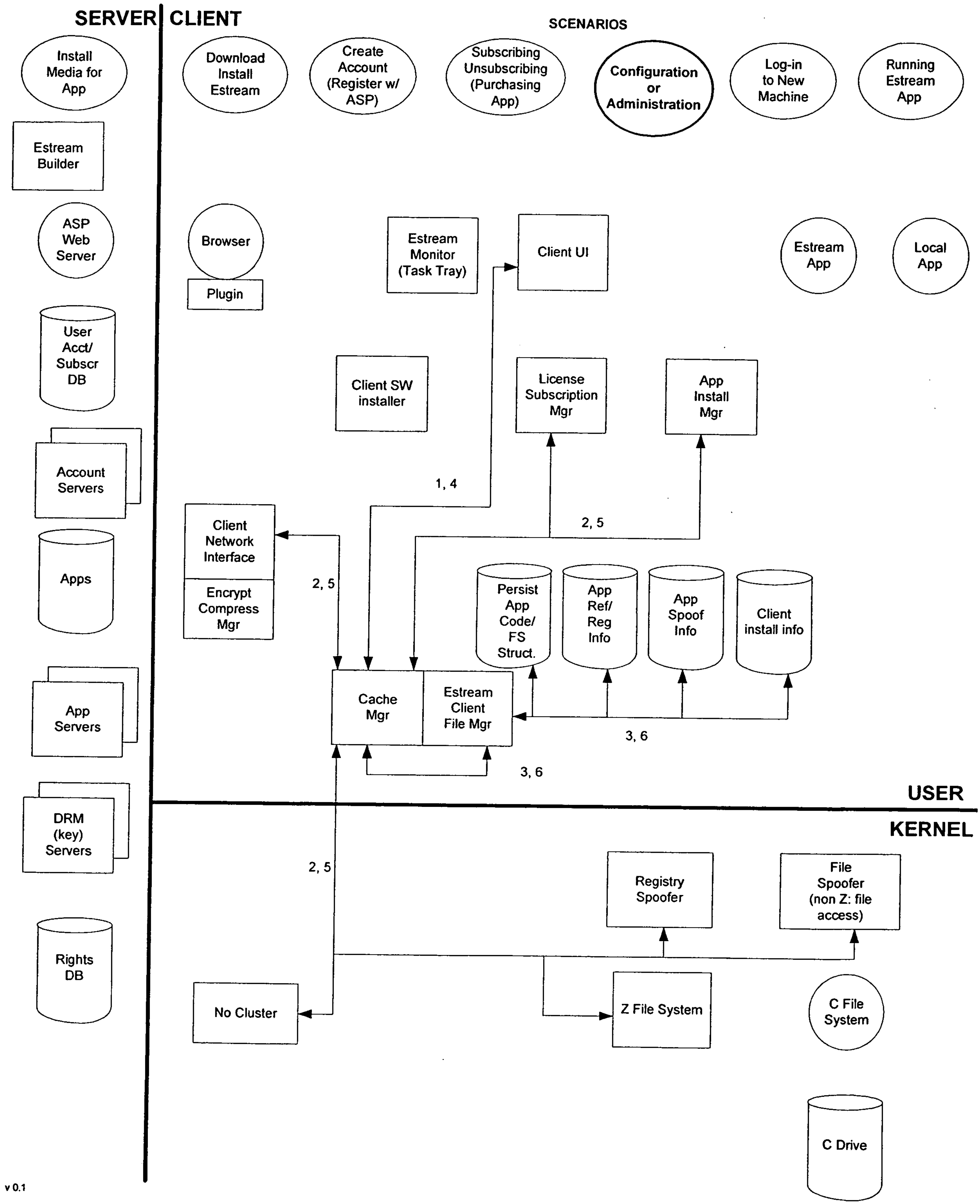
Cross-component testing plans

Cross-component testing of the client NW includes integration testing with the EMS, the LSM, and the prefetcher. Testing with the EMS can be performed in a manner similar to unit testing in conjunction with a specially written server. Testing with the LSM or pre-

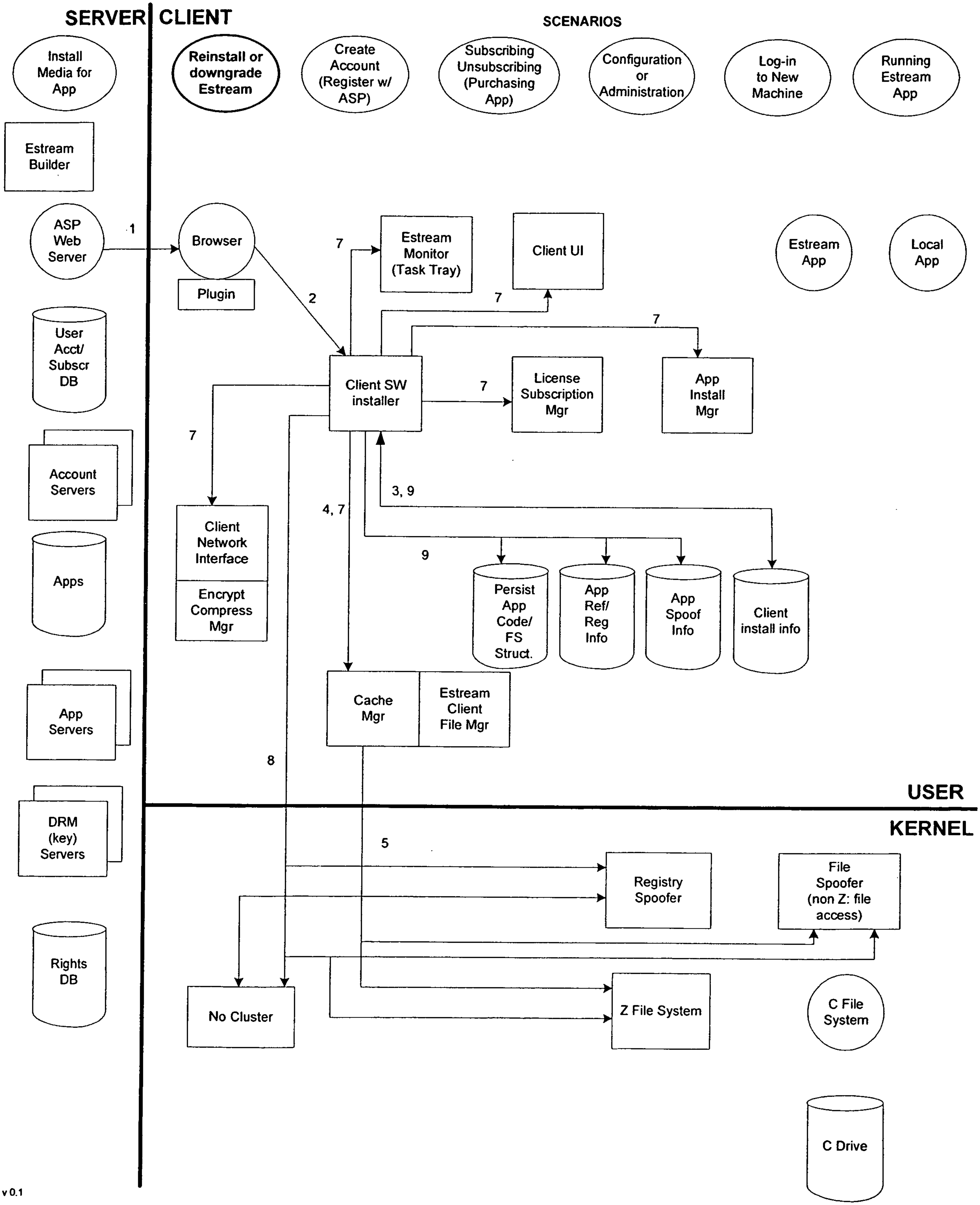
fetcher can be performed in isolation by writing drivers for either the LSM or prefetcher, and using a dummy or real EMS. I'm not sure if this sort of testing is worth the effort to write the appropriate harnesses. Verifying the output of such a combined system is certainly trickier than testing any component in isolation.

Open Issues

eStream Client Configuration



Scenario 5: eStream client SW downgrade/reinstall



v 0.1

eStream 1.0 Cache Manager Low Level Design

Version 1.4

[redacted]

Omnishift Technologies, Inc.
Company Confidential

Functionality

The eStream cache manager implements much of the client-side functionality for handling the eStream file system. The cache manager handles all file system requests made by the operating system by reading information from the cache or by passing the requests along to the profiling and prefetching component to fetch missing data from the network.

The cache manager will initially be implemented in user space, but it may be useful to migrate it to the kernel for improved performance. In user space, it will be part of the eStream client process. In the kernel, it will probably be a device driver distinct from the eStream file system driver.

The cache manager manages the on-disk cache of file system data, and the in-memory data structures for managing this cache. It does not manage prefetching of data from the server; that is the role of the eStream Profiling and Fetching (EPF) component. A separate networking component handles the network traffic. This component will also be described separately.

Since there is no overall discussion of the client architecture at a more detailed level than the high level design, this document will cover that as well.

Multiple cache page files will be supported. Each cache page file may be up to 2 GB in size. Different cache files may reside on different or the same logical disk (i.e. Windows drive letter.)

Data type definitions

An application ID uniquely identifies an eStream application. Just what constitutes "one" eStream application is not entirely defined, but different "builds" of the "same" app will be considered different eStream applications. For example, the Chinese-language version of Office is a different eStream application than the English-language version.

```
typedef uint128 ApplicationID;
```

The eStream page number is the data type used to describe a page number within a particular file. Note that this is a page offset, not a byte offset. For eStream 1.0, the cache manager will only support 2 GB cache files.

```
typedef uint32 EStreamPageNumber;
```

The fileId is used to uniquely identify a file within the universe of all eStream files across all eStream applications.

```
typedef struct {
     ApplicationID App,
     int32 File
} fileId;
```

The eStream page size is the fundamental size for eStream requests. This size is in bytes.

```
#define ESTREAM_PAGE_SIZE 4096
```

The eStream file system uses the file time format of the Windows operating system. If the client runs on a system with a different native time format, the client software will be responsible for translating between the native format and the eStream format. The Windows data format is a 64-bit counter of the number of 100-nanosecond periods since January 1, 1601.

EStream metadata is the file information supported by the eStream file system. This metadata is independent of the client or server operating system.

```
typedef struct
{
     uint64 CreationTime;
     uint64 AccessTime;
     uint32 FileSize;
     uint32 FileSystemAttributes;
     uint32 EStreamAttributes;
} Metadata;
```

The eStream inode contains the layout of a file in the cache. Each inode has the following structure:

```
typedef struct
{
     FileId Id; /* ID of this file; search parent for
name*/
     Metadata Metadata;
     FileID Parent; /* parent directory's file id */
     uint32 NumPages;
     PageInfo *Pages;
} EStreamInode;
```

The PageInfo array is variable sized. There is one entry in the pages array for each page in the file (not for each page cached, since we need to know whether the pages are present or not...) Note that the inode is only used in the "robust" implementation.

```
typedef struct
{
     EStreamPageNumber CachePageNumber;
     PageStatus Status;
     unsigned char Priority;
     PageChecksum Checksum;
} PageInfo;
```

The page number doesn't require the 32 bits, since pages are 4096 bytes long. The extra bits will be used to encode which cache file this page resides in. The priority field is a number representing this page's priority for being kicked out of the cache. How exactly this field is used hasn't yet been determined. The checksum is a (fast) page checksum that can be used to validate the contents of this page. Note that it will be useful to have a slower, more effective checksum for development and a faster (but less thorough) checksum for deployment.

The page status is an enumeration for the page's locking status (these are described in more detail later:

```
typedef enum
{
     PS_INVALID,
     PS_CLEAN_UNLOCKED,
     PS_CLEAN_LOCKED,
     PS_DIRTY_UNLOCKED,
     PS_DIRTY_LOCKED,
     PS_IN_FLIGHT
} PageStatus;
```

Note that this describes the layout of the tables in memory; how these data structures are represented on disk is described later.

The EFSD file handle is a small integer passed between the EFSD and the ECM. This is used opaquely by the EFSD and is used as an index into an open file table by the ECM.

```
typedef uint32 EFSDFileHandle;
```

The ECM request type specifies the request type to the rest of the system. Note that some "requests" are used to inform the prefetcher about the events handled solely by the ECM, and do not actually request that any particular action be taken by the prefetcher.

```
typedef enum
{
     ERT_READ,
     ERT_WRITE,
     ERT_READ_HIT,
```

```
    ERT_WRITE_HIT
} ECMRequestType;
```

The ECM request is a request descriptor that is used in various lists within the cache manager. These lists are doubly-linked, circular lists.

```
typedef struct _ECMRequest
{
    uint32 RequestID; /* same as EFSD request id */
    ECMRequestType RequestType;
    union {} Parameters; /* union of all parameters*/
    struct _ECMRequest *next;
    struct _ECMRequest *prev;
} ECMRequest;
```

The cache manager must maintain an array of files that have currently been opened by the EFSD. This array will be statically allocated. This will put a limit on the number of files that may be opened concurrently on the eStream file system. The elements of the array are the following:

```
typedef struct
{
    uint32 Valid;
    fileId File;
    HANDLE OpenFile; /* for simple implementation */
    eStreamInode *Inode; /* for robust implementation */
} OpenFileInfo;
```

The cache manager maintains a hash table containing information about each application that currently has open files. The hash table is indexed by app ID, and contains the following active app information records:

```
typedef struct
{
    AppID App; /* identity of this app */
    uint32 OpenFiles; /* # of open files */
    uint32 HaveAccessToken; /* boolean */
} ActiveAppInfo;
```

The ECM will use this table to quickly determine whether it should continue processing a request it gets from the EFSD, or if the request should be passed to the LSM to ensure that an access token is available. See the section below on ECM-LSM interaction for more details.

The LSM uses the access token state to specify a state for an access token. Right now, we only plan to support valid and invalid, but it may be interesting in the future to allow already opened files to be read, but no new files to be opened.

```
typedef enum
{
     ATS_INVALID,
     ATS_VALID,
     ATS_VALID_NO_OPEN
} AppTokenState;
```

Interface definitions

The ECM exports the following interfaces for operating on the cache. They may be called by the cache manager, prefetcher, or networking component. (Not all components are expected to call all interfaces; see each interface description for more details.)

Note that the cache interfaces are defined at a very high level as the actions that may be performed on the cache by the components, such as enqueuing a new request. They have been defined this way so that these intrinsic operations can be implemented correctly once and limit the possibility that an individual component will not perform proper actions.

ECMReservePage

eStreamStatus ECMReservePage(
    IN fileId *File,*
    IN EStreamPageNumber *Page,*
    IN ECMRequest **Request*
);

ECMReservePage reserves a page in the cache for a request. This interface is called by the prefetching component, and will send a request to the network component. Logically, this interface reserves an empty cache page for this request (if one is available), puts this request on the "in flight" queue, and calls on the network to request the page (unless it is already in flight.)

ECMIsPageInCache

    eStreamStatus ECMIsPageInCache(
    IN fileId *File,*
    IN EStreamPageNumber *Page*
);

ECMIsPageInCache returns TRUE if the specified block is in the cache, and FALSE otherwise. It is used by the EPF to determine if it should prefetch a block; normally, the EPF would choose not to prefetch something that is already in the cache. Note that it would be a good idea for the prefetcher to adjust the priority of a page that it thinks it wants to prefetch, so that they are less likely to be evicted from the cache before they are needed.

ECMDeplanePage

eStreamStatus ECMDeplanePage(

IN fileId *File,*
IN EStreamPageNumber *Page,*
IN char *Buffer[ESTREAM_PAGE_SIZE]*
);
ECMDeplanePage performs all the necessary actions for writing a page coming off the network into the cache and back to the EFSD. This consists of copying the page into the cache, remove all pending requests for this page from the in flight list, marking the page as clean/unlocked, and returning the page to the EFSD for each in flight request.

ECMReadPage

eStreamStatus ECMReadPage(
IN fileId *File,*
IN EStreamPageNumber *Page,*
IN ECMRequest **Request*
);
ECMReadPage performs all the necessary actions for attempting a page read from the cache. The cache is checked to see if it contains the page; if so, the page is copied to the buffer, the EPF is notified of the hit, and appropriate status is returned. Otherwise, this page is put on the queue for requests pending to the prefetching component, and appropriate status is returned.

ECMWritePage

eStreamStatus ECMWritePage(
IN fileId *File,*
IN EStreamPageNumber *Page,*
IN ECMRequest **Request*
);
ECMWritePage performs all the necessary actions for attempting to write a page in the cache. Note that this could be somewhat more complex than a read, because a partial write to a page might necessitate reading the page from the server before writing the partial page to the cache.

The following interfaces are the abstract interfaces that the ECM will use to communicate with the EFSD. Hiding the EFSD's raw DeviceIoControls behind these interfaces will help make porting the ECM into the kernel easier, should we decide to do that.

ECMSetTokenState

eStreamStatus ECMSetTokenState(
IN AppId *App,*
IN AppTokenState *State*
);
ECMSetTokenState is called by the LSM to indicate to the ECM that a token has become available or has expired. The main effect of this interface is to update the state of the specified application in the active app table. See the ECM-LSM interaction below for more details.

ECMGetCacheInfo

eStreamStatus ECMGetCacheInfo(
  OUT UNICODE_STRING *Location,*
  OUT uint32 **CurrentSize,*
  OUT uint32 **MaximumSize*
);

ECMGetCacheInfo is called by the client user interface to find out where the ECM cache is located and its current and maximum size. Location is an absolute path name of the cache file.

ECMSetCacheInfo

eStreamStatus ECMSetCacheInfo(
  IN UNICODE_STRING *Location,*
  IN uint32 *MaximumSize*
);

ECMSetCacheInfo is called by the user interface when a new cache location or size has been requested. Note that the cache manager may only begin using the new cache information after a restart of the client software (which may only occur on client machine reboot.) The client UI will call this interface when it wants to make a change; the ECM is responsible for actually resizing the cache and making any changes necessary to persistent storage (i.e. the registry).

EFSDGetRequest

eStreamStatus EFSDGetRequest(
  OUT EStreamRequest ***Request*
);

EFSDGetRequest reads the next request from the EFSD, including any parameters that need to be passed. This may involve one or more DeviceIoControl calls to the EFSD. **EFSDGetNextRequest** is responsible for allocating memory for this request, and an **EFSDCompleteRequest** call will be repsonsible for deallocating the memory.

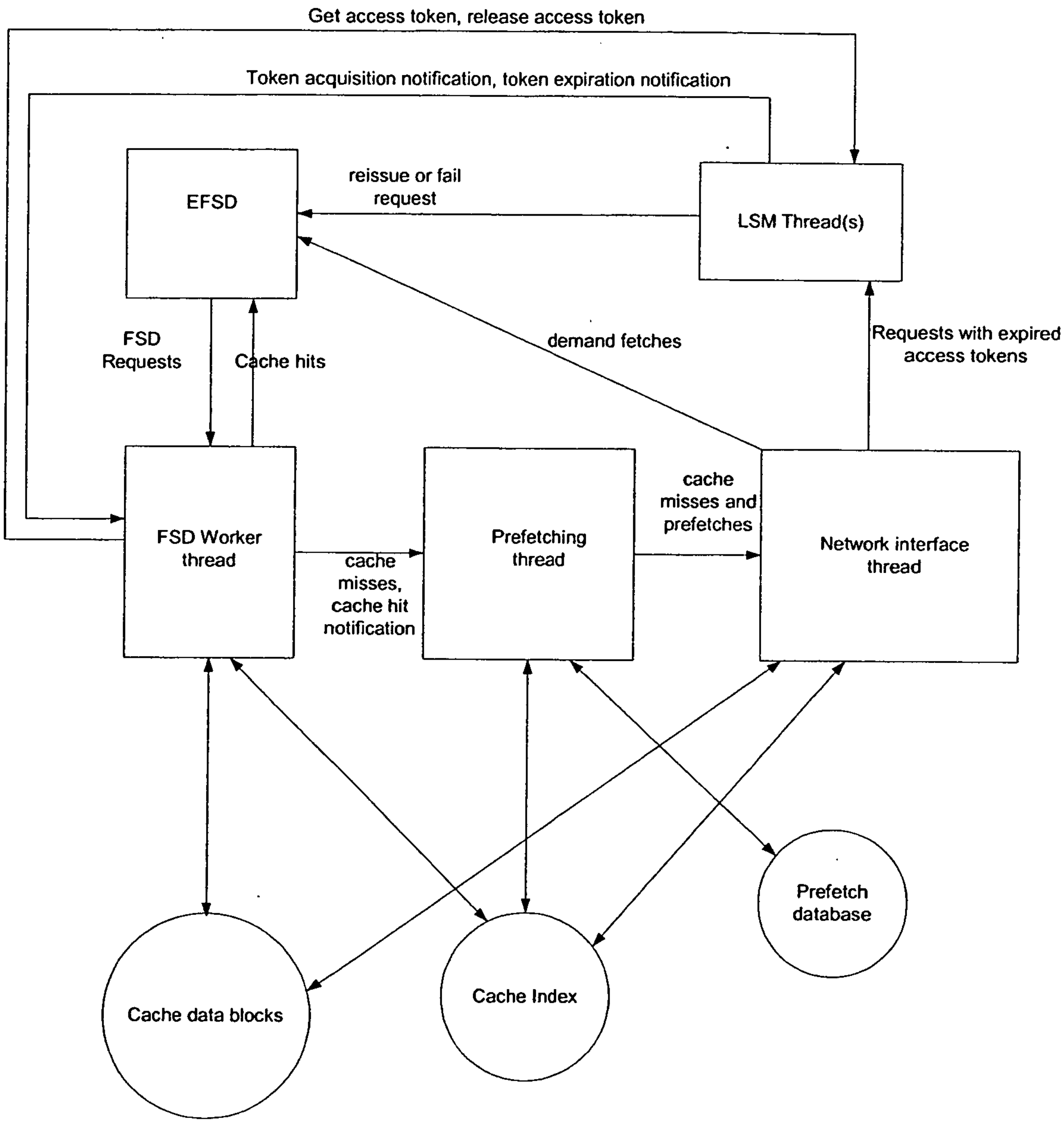
EFSDCompleteRequest

eStreamStatus EFSDCompleteRequest(
  IN EStreamRequest **Request,*
  IN ECMErrorCode *Status*
);

EFSDCompleteRequest will be called for each request that is received by the ECM via EFSDGetRequest. *status* indicates the completion status for this request, and may indicate success, a retry, or a particular failure condition. Non-persistent errors will be handled by the ECM internally or by requesting a retry of a particular request. Errors reported to the EFSD will be propagated up the file system stack.

Overall Client Architecture

The eStream client will have various types of threads in order to perform its work. The basic architecture is illustrated by the following diagram.



The FSD worker thread will pull requests from the FSD. It will return data for requests that can be satisfied immediately. Any request that requires information that is not currently in the cache will be put on a queue for the prefetching thread to handle.

The profiler will receive all cache misses from the FSD worker thread. Using its own data structures (which may include information about recent cache misses in addition to information about general prefetch patterns), it will decide which blocks it should prefetch. Demand fetch and prefetch requests are sent to the network component. The

only way demand fetches and prefetches are treated differently by the network component is that demand fetches are sent to the EFSD while prefetches are not.

The network thread will manage open connections to app servers and retry requests that time out. When data comes back from the network, the network thread will copy the returned buffer into the cache and to the FSD, if the request was a demand miss.

The cache manager consists of the EFSD worker thread and the APIs to access the cache index, the data blocks, and various queues used by threads in the client.

Not shown on the diagram is an error thread. This thread is responsible for calling the client UI module indicating appropriate error messages and waiting for the user's input. When any component decides that it has an error condition that requires user input, it calls **ECMReportError** with the request and an appropriate error condition, which will be enqueued for the error thread to handle. For example, when the network interface times out reading a page from an application server enough times, it will call **ECMReportError**. When the error thread gets to this request in the queue, it will ask the user if he wants to wait until the app server is available or allow the application to terminate.

ECM-LSM Interaction
The ECM-LSM interaction is a relatively simple one. The LSM notifies the ECM when it first receives an access token and when its access token expires. It does this via the **ECMSetTokenState** interface. The ECM keeps track of each application that has had files open, and whether or not we have an access token for each of these apps.

App ID	# of open files	Have access token?

Note that the LSM need not notify the ECM of mundane events like renewals as long as some token is valid. Also, the ECM does not keep track of the token itself, just whether or not we have a valid one. An additional nicety of this approach is that we could allow the ECM to satisfy requests out of the cache as if we have an access token, without actually having one.

When it receives a request, the ECM checks its table to determine if an access token is available. If it is, it handles the request as normal. If not, it asks the LSM to acquire an access token via **LSMGetAccessToken**. The LSM may return that it has a token, in which case the ECM will continue to process the request, or the LSM may say it doesn't have a token, in which case the LSM takes ownership of the request and will reissue the request when the access token is available.

When the number of open files drops from 1 to 0, the ECM will mark the token as invalid in its table and call **LSMReleaseToken**. The LSM may choose not to renew access tokens that have been released.

Component design

Two cache organizations will be presented. One is suitable for a quick implementation but doesn't lend itself particularly well to high performance or easy manageability; the other will be more difficult to implement but should provide better performance. I will first describe some data structures that are shared by both designs, then go into the specifics of each design.

Common Data Structures and Algorithms

Certain request lists are common to both cache organizations. One is a queue between the FSD worker thread and the prefetching thread for demand fetches that have not yet been seen by the prefetcher. The other is a list of all requests for pages that are "in flight." Requests from the in flight list are removed when they have been satisfied. The in flight list is unsorted and searched whenever a request comes back for requests that match the returned page. If the performance of this data structure becomes an issue, we will change its organization for faster lookup.

Both request lists use the request data structure described above.

The ECM will maintain an array of files currently opened by the EFSD. On file opens, an empty location in this table will be allocated for the newly opened file, and the index to that entry returned as the file handle. (Note that the way the interface between the ECM and the EFSD is defined, it is an error to open an already opened file. The cache manager will have to detect such cases and report an error, but it will not keep a reference count of the number of opens on each file.) This mechanism will allow the ECM to keep track of the volumes that currently have opened files as well as abstracting the client/server file ids away from the kernel driver. (This might allow us to update the client/server protocol without rewriting the EFSD.)

Easier Implementation

The cache will be implemented as a directory tree on the user's hard drive that parallels the eStream file system. Each file will contain a header and an array of status bytes in addition to the data blocks that the file contains. The array of status bytes has one byte for each page in the file. Each byte indicates the current status of that page in the file. (Pages have several different states, so a simple bit per page is not sufficient.) Each file will thus look like

Header
Page Status Bytes
File contents page 0

File contents page 1
...

The header is defined as:

```
typedef struct
{
uint32 magicCookie;
uint32 headerLength; /* Length of this header, in bytes */
fileId fileid; /* for sanity checking */
uint32 length; /* Length of the file, in bytes */
uint32 firstPage; /* Offset to the first page in the file
*/
Metadata metadata;
} ECMCacheFileHeader;
```

The page status bytes begin immediately following the header, and this area is padded with zeros to a page boundary. The first page of the file's contents (and thus each following page of file contents) will therefore begin on a page boundary.

Note that one issue with this design is that files that approach the file size limit of the underlying file system cannot be represented, due to the overhead with the header and bitmap. If this design is used solely for early engineering efforts, then this limitation is acceptable. If we have to work around this limitation, one way to do it is to make the headers and page status bytes reside in a separate file or files.

Directory contents would reside in server format in a file named "Directory" inside of the directory whose contents they represent (with the addition of the header and status bytes as described above for ordinary files). For example, z:\Program Files\Microsoft Office would reside in c:\Cache\Program Files\Microsoft Office\Directory. This has the drawback of creating special file names that can't be used by files in the eStream volume, but again, for an early engineering implementation, this is an acceptable limitation.

Another issue with deploying this implementation is that it is trivial to reverse-engineer this file format and copy files directly from the cache.

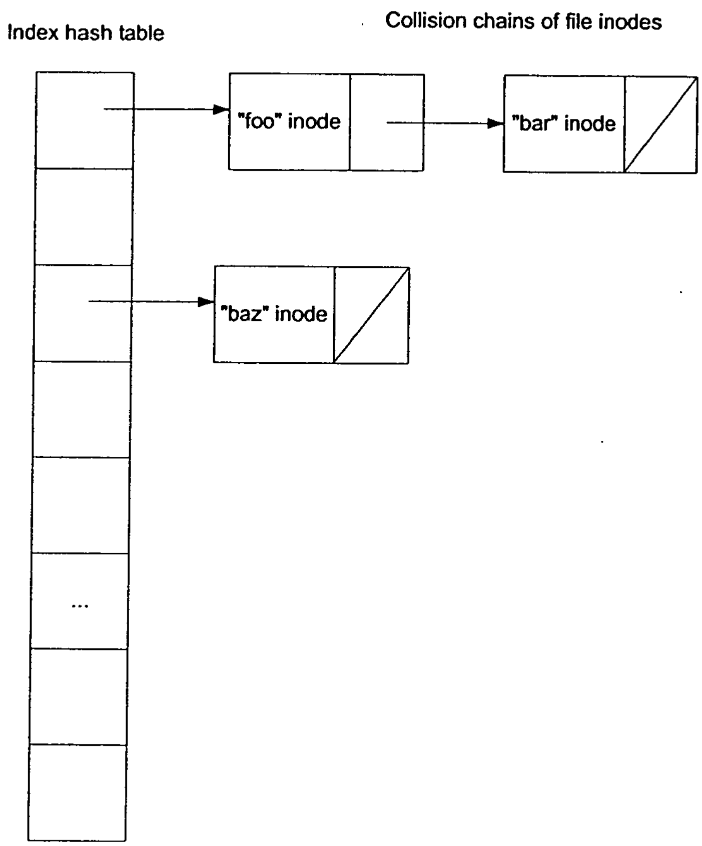
Robust Implementation

The cache will be organized into an index file and one or more cache data files. Multiple data files may be necessary as we may wish to allow the cache to grow larger than the 2 GB file size limit (for some native file systems) or to span multiple drive letters on the client. The data files will only contain pages of file content. These pages will be aligned on page boundaries. The index file contains all the information needed to locate file pages, and is contained in a separate file for simplicity.

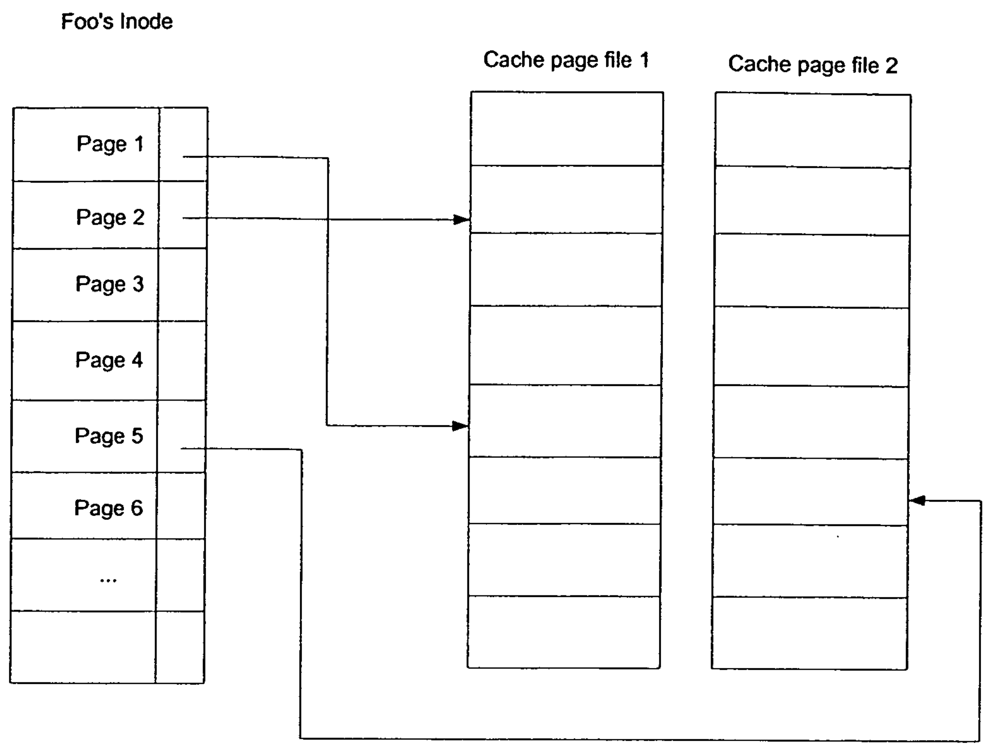
Page and index files must reside on a local disk (rather than a network disk) and cannot be shared by multiple clients.

Each file with any pages currently resident in cache will have a data structure containing information about that file, including its file id, the file id of the directory containing it, the file's metadata, and the map for finding the file's data blocks. This data structure is very similar to the inode of a traditional file system, and will be referred to as the eStream inode. A naive implementation of the inode is described above; no doubt, we will want to reorganize this data structure for more compact representation and better performance. Note that one requirement of the inode is that it contain a status field for each page in the file. One character is sufficient for this status; whether or not we can make do with fewer than 8 bits is an open question.

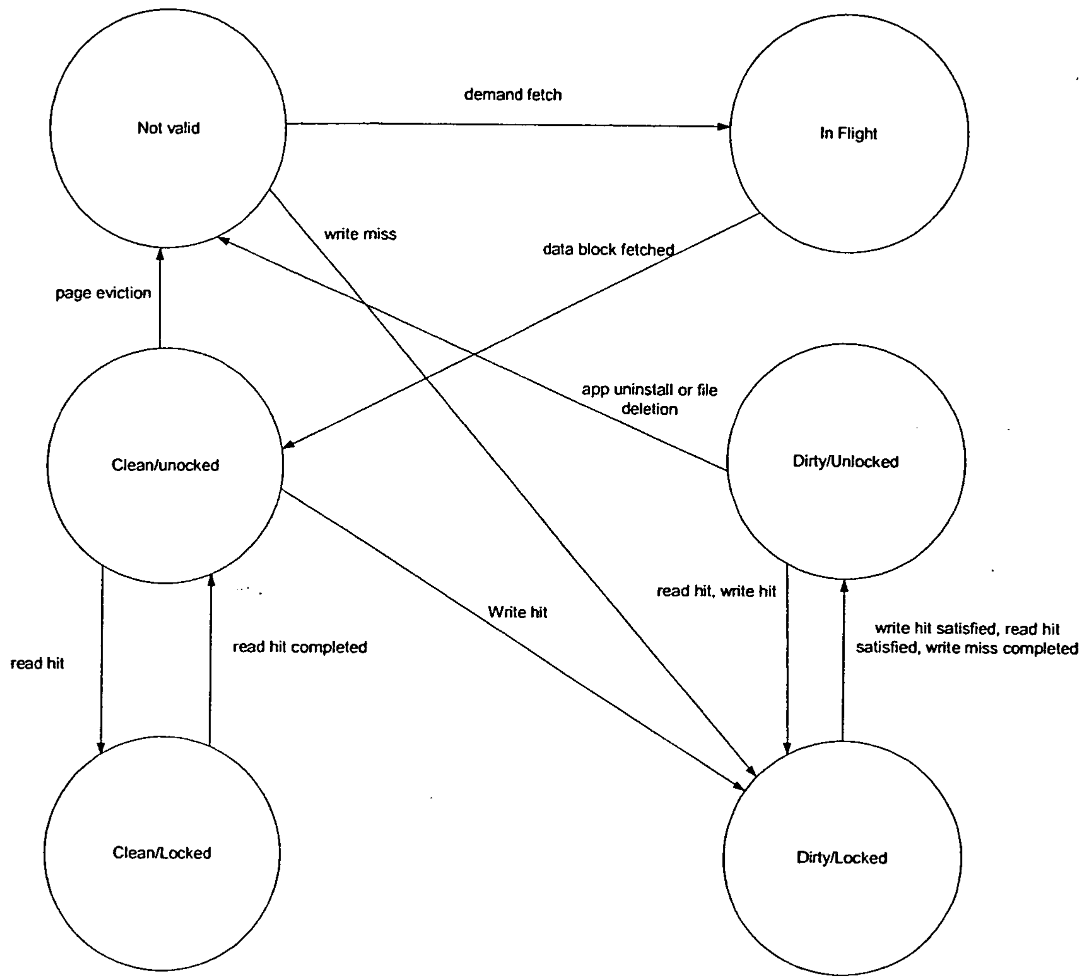
A hash table will be used to map file IDs to file inodes.



The inode contains pointers to each block's location in one or more cache page files:



To prevent race conditions, a single lock controls access to both the hash index and the linked list of requests that are pending network access. Individual pages in the cache may be locked for read or write access. Since each page's status is in the index, the index must be locked order to lock a page for reading or writing. The page states are controlled by the following state machine:



The dirty/clean distinction is between those pages that we have written locally (and thus cannot evict from the cache) and those pages that we haven't written (and thus can be refetched from the server).

A page would be locked while it was being read or written for copying to the file system driver. The operation may thus proceed with the index unlocked, without the possibility of page eviction while a copy is still in progress. The FSD worker thread is the only thread that reads or writes pages from the cache, so it's the only thread that can lock or unlock these pages. The in flight state is only for pages that are currently being fetched, either as a demand fetch or as a prefetch. The prefetching thread is the only thread that will put pages into this state, and only the networking thread will move pages from in flight to unlocked.

A list will be maintained of all "in-flight" requests. A single lock will control access to both this list and the cache index, so there are no race conditions between items being put on this list and data coming off the network. When the FSD worker thread gets a request, it acquires the index lock and looks at the status of the page. If the page is clean or dirty but unlocked, it will lock the page and copy it to the FSD. If the page is invalid, then this is a demand fetch, and the request is forwarded to the prefetcher. If the page is marked in

flight, then this is either a second request for an outstanding demand fetch, or it is a request for an in flight page. Either way, while this thread still holds the index lock, this request will be inserted into the list of in-flight requests. Race conditions might occur because the FSD might make multiple demand reads of the same page, or it may make a demand read to a page that is already in flight due to a prefetch.

Reading requested pages off the network and writing them to the cache (and to the file system driver, if necessary), are where this race condition comes up. We need to ensure that a request for a page that has arrived does not end up in the list of "in flight" requests. The solution is the following: When a data page comes back from the server, the networking component acquires the index lock to find the cache location of this incoming page. If the page is not marked in flight in the cache, this is a bug. (Of course, this is a relatively benign bug, and the NW component could just ignore the page.) The networking thread leaves the page as marked in flight, however, and unlocks the index. It writes the incoming page into the proper location, but it saves the in-memory copy of the page. It then reacquires the index lock, marks the page as clean/unlocked (since it's now in its final location in the cache), removes each request in the in-flight list for this page, then releases the lock. (Any further requests for the same page will find the page clean/unlocked, so the FSD worker thread will be able to satisfy these requests directly.) The networking component then proceeds to satisfy all of the requests it pulled off the in-flight list by using the copy of the page that it saved in memory. This way, it doesn't have to lock the index the entire time it is sending completed requests to the FSD.

Each of these complex scenarios is captured in the cache file's API's. As long as these are implemented correctly, other components don't need to worry about the exact sequence of operations that needs to occur.

Free Space Management
Free pages will be maintained as a free list in memory and as a bitmap on disk. The free list will be built from the bitmap on eStream client software startup. Access to the free list will be controlled by the same lock controlling access to the index.

Evicting Cache Pages
Individual cache pages may be evicted. There is an 8-bit field in the index for each page's importance. Initially, we will implement a random page replacement policy. Later, we will use this page importance field in an unspecified way to replace pages in such a way as to maximize interactive user performance and minimize application server load. Only clean/unlocked pages may be evicted. Pages that are evicted will eventually be put on the free list. Page eviction will only happen at "garbage collection" time. See "crash resilience and garbage collection," below.

Handling Cache Size
Growing the cache should not be an issue. The cache manipulation routines must know the overall size of the cache, in pages. Increasing the size of the cache on the fly should be a relatively straightforward process, as we merely need to lengthen the cache file(s) and add the new pages to the free page list.

Unfortunately, shrinking the cache is a much more difficult operation, since it potentially involves moving around pages that might currently be in use for paging operations or be in flight from the network. Changing the cache around at runtime is both difficult to implement correctly and a performance problem. The current plan is to support shrinking the cache only at eStream client software startup. The maximum allowed size of the cache will be stored in the Registry. On eStream client software startup, the current size of the cache will be compared against the allowed size specified in the registry; if it is larger than the maximum size specified in the registry, then the size of the cache will be reduced by evicting files and compacting the freed space. A request by the user to reduce the size of the cache will take effect the next time the client software starts.

Note that files that the user writes to the z: drive are not considered candidates for eviction (unless the file is explicitly deleted.) This means that the user's on-disk cache may in fact grow to be larger than the limit they specify.

Also note that at least one free page (not used by user-written files) is required for the file system to make forward progress. We also may want to require some minimal amount of cache before eStream will even run. Thus the maximum cache size specified by the user should be considered a "soft limit." There would be a "hard" minimum amount of space equal to the number of pages required to store the files written by the user on the z: drive plus a small amount of cache we designate just for running eStream. If this hard minimum is greater than the soft maximum specified by the user, the hard minimum would win. I would recommend preallocating and non-zero filling the file on disk so that we know that the space is available.

Crash Resilience and Garbage Collection
In order to provide crash resilience, the index will be periodically checkpointed to disk. Note that allocating blocks does not cause problems if the index is not updated. However, we cannot reuse a page's storage until that page has been marked free on disk.

The solution to this problem is to periodically garbage-collect the cache (if it is nearly full), and writing the index to disk. The cache manager will alternate between writing two cache index files. The index file will have a marker at the end that indicates that it has been successfully written and a time stamp, and on startup the ECM will use the latest, fully written index.

Data blocks will always be written directly to the cache page files. These files must be flushed before writing the index.

Garbage collection involves the following steps:
- lock the index
- copy the free list
- choose blocks in the cache to free, and make a list containing just the newly freed blocks. Mark these blocks as invalid in the file's inodes, but don't put them on the free list (yet)

- make a copy of the index
- unlock the index
- merge the list of newly freed blocks with the copy of the free list
- flush all cache page files
- write the new, merged free list (as a bitmap) and index to disk
- lock the index
- add the newly freed blocks to the free list
- unlock the index
- free any allocated data structures

Index File Contents

The index file contains the following items:
- List of cache block files, with their sizes
- Free block bitmap, per cache block file
- Inodes for all files; may be stored hashed or may be rehashed on startup.

Testing design

Unit testing plans

Cache file manipulation routines can be tested in isolation. We will write a standalone harness that exercises the functionality of the cache file manipulation routines by performing cache level operations directly. A multithreaded unit test for the cache manipulation routines would be ideal, so we can test the correctness and performance of our locking strategy without the need to build the entire cache manager.

Each "thread" of execution described by this document can be separately tested by creating a testing harness providing that thread inputs and monitoring its outputs. Replacements for the EFSD interfaces can be very effective here.

Stress testing plans

An interesting stress test for the cache manager is if it can work correctly with very small caches, even all the way down to 1 page. (Or at least, a cache with all pages but one marked as dirty.)

The cache manager will be able to operate in "verify mode," where requests that hit in the cache will still be sent to the server, and the pages returned by the server will be compared with the cached page's contents.

The cache manager will support multiple different page checksum algorithms. We can use a fast algorithm for deployment while using a more rigorous one in development. This also has the benefit of allowing us to test the performance impact of various checksum algorithms.

The cache manager will have the ability to verify the integrity of the cache index and free page bitmap. In particular, it will have the ability to determine taht no pages are allocated to more than one file in the file system, and that each page belongs to a file or is on the free list.

Stress testing for the ECM will include crash testing.

Cache manager testing will include resizing the cache.

Coverage testing plans

Cross-component testing plans

We can build a "cache only" file system by not using the prefetching and network components. This allows us to test the EFSD in conjunction with the cache manager without involving the prefetcher or the network component.

Early implementation of the client will likely involve a null prefetcher that does no prefetching.

We can use the testing harness for the cache manager that doesn't use the EFSD to drive the cache manager in conjunction with the prefetcher and network component. This allows us to test the combination of these components without driving it with the live file system driver.

Upgrading/Supportability/Deployment design

The client user-mode software and device drivers are packaged separately. (I.e. the client executable and the drivers are separate files on the disk.) This leads to the possibility of a "partial" upgrade that results in inconsistent versions of the drivers and client user-mode software. The drivers should support an interface that returns the version number of the driver, or of the interfaces provided by the driver. This will help the client software to recognize situations where it should tell the user to reinstall the client software and not result in bad system behavior.

Most (all?) on-disk data files should have file headers containing at a minimum a magic cookie and the file format version number. This will help us with upgrades in the future.

Open Issues

We need to address what happens when a fetch is requested and no empty space can be found in the cache. The prefetcher should probably block until such time as space is

made available for this request. While operating with very small amounts of cache will obviously cause bad performance, it should not result in a deadlock.

eStream Cache Manager Straw Man Proposal

Version 0.2

[redacted]

Purpose

The purpose of this document is to serve as the basis for the design of the eStream Cache Manager. As a straw man, this document is meant to serve as the basis for discussion, and anything here is subject to change. Assuming there are no major concerns with this document, I will proceed with producing a low level design for the cache manager.

Requirements in Brief

Support > 2GB client cache, possibly across multiple drives
Provide some level of protection against piracy, via both the file system and the cache
Fast lookup for what is in the cache and where to find it
Support automatic and user-specified cache size policies

As far as cache size goes, I think that it is reasonable for eStream 1.0 for the cache to be limited to one disk partition and 2GB of space, but the design should allow for very large caches (spanning more than one file and possibly more than one drive letter.) Note that if the cache is greater than 2GB in size, it cannot be mapped into the address space of a single process under NT/2000 on x86.

Cache Organization

The cache will be contained in 2 or more files. One file will contain the cache indices, and one or more files will contain the data blocks for cached files. (More than one cache data file may be required if the cache is larger than the largest file allowed on the native file system.) This allows us to keep the cache index file memory mapped and only map the data file(s) if there is enough memory space to do so.

Data Blocks

The cache data file will contain data pages frome the file system 4k in size.

Data will be stored in the cache uncompressed to allow easy page retrieval.

Cache Index

The cache index will be a b-tree. The key for the lookup will be the file id and page number requested. Keys in the b-tree are the set { volume #, file #, starting page, # of pages }. A lookup will succeed when the volume number and file number match, and the requested page is in the range from starting page to starting page + # of pages. The data stored for that key will be the offset into the cache for the beginning of the run. As is described in the file system proposal, the file number and starting pages are each 32 bits long. I propose making the starting page a 48 bit number and the number of pages a 16 bit number. This allows us to have a very large total cache and reasonable sized runs of contiguous pages in the cache.

Free space in the cache will have to be managed. Free blocks can be placed into a specially identified "free space file" in the index. Some auxiliary data structures may be convenient to make searching for a region of free space of a particular size.

Metadata for a file would be stored in the cache. It would be indexed by page number -1 in the index.

Cache Replacement Policy
For simplicity, I propose that the cache manager evict entire files from the cache when it decides that it needs to clear room in the cache. (Of course, any fragmentary file that is in the cache can be evicted.) We should implement LRU for cache replacement, so we will evict files for apps that have not been run recently.

One Cache Per System
Administrator priviledges are required to install eStream. While various users on a system might have conflicting desires about eStream configuration, such as the size of the cache, I think that it is reasonable to have a policy where the adminstrator controls the setup of the eStream client. By limiting the cache to one per system, we eliminate any ambiguity about cache use in a multiuser environment.

Profiling and Prefetching
Profiling and prefetching have been broken out as a separate component in the client. It will be described elsewhere. It is expected that while the profiler/prefetcher will want access to the cache data structures (i.e. it wants to know what's already in the cache), the logic associated with prefetching is not logically tied to the cache manager, and should thus be separated.

Future Directions
Compression of the cache could potentially be a big win. We could provide cache compression similar to the way that NTFS provides file compression - we compress some number of blocks at a time (e.g. 16) and only store the compressed data when it saves at least one block of storage. Caching of data on disk can sometimes be a performance win, since decompressing the data can be faster than trasferring it on disk if the disk is slow enough.

eStream File System Driver Low Level Design

version 1.4
Curt Wohlgemuth
[redacted]

Functionality

The eStream Windows NT/2000 File System Driver (EFSD) is a kernel-mode file system driver to which file requests will be forwarded by the NT I/O Manager. It is the point of contact for users to access files on an eStream server. It works with the NT File Cache Manager to insure that kernel file caching is available for eStreamed files.

The Windows 98 EFSD is almost certainly to be very different from the driver for WNT and Win2K, and will not be described here.

In this document, I'm assuming that the EFSD communicates closely with the eStream cache manager (ECM) to perform the various file system requests. There may in fact be several components—if for example the ECM is broken into sub-components. Also, this document assumes that the ECM is in user mode; if this ends up in kernel mode, we will need significant changes to the interfaces to it.

Data type definitions

File handle

A file handle passed between the EFSD and the ECM is defined by the ECM:

```
typedef uint32 EFSDFileHandle;
```

Names

All file and directory names will be passed as counted Unicode strings, basically as defined by the NT header files. Note, however, that in NT the Buffer field is a pointer; for our purposes in communicating with the ECM, it's a NULL-terminated variable length array:

```
typedef struct _UNICODE_STRING {
   USHORT Length;
   USHORT MaximumLength;
   USHORT Buffer[1]; // NULL-terminated, 2-byte
                     // characters
} UNICODE_STRING;
```

Time stamps

The NT standard time format is a signed 8-byte integer representing the number of 100-nanosecond intervals since January 1, 1601. These time stamps will be tracked for files and directories:

- Creation time
- Modification time

File attributes

File attributes are contained in an unsigned 4-byte integer. This subset of attributes from Windows NT will be supported:

```
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_TEMPORARY
```

These attributes are not supported:

```
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_DEVICE
FILE_ATTRIBUTE_SPARSE_FILE
FILE_ATTRIBUTE_REPARSE_POINT
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
FILE_ATTRIBUTE_ENCRYPTED
```

File size

File size will be represented as a 4-byte unsigned integer. Since sparse files are not supported, there will only be one file size passed between the ECM and the EFSD.

Metadata

This structure is defined to pass file and directory metadata between the EFSD and the ECM:

```
typedef struct {  // 24 bytes, 4-byte aligned
  int64 CreateTime;
  int64 ModifyTime;
  uint32 FileSize;
  uint32 Attributes;
```

```
} Metadata;
```

Interface definitions

The EFSD is called by several different components, including

- the NT Executive (I/O Manager, Virtual Memory Manager), for standard file system requests
- the ECM, for these same file system requests, and to invalidate cached pages for coherency
- the client start software, to start and stop the EFSD

The EFSD supports standard FSD interfaces to the NT Executive modules; not all possible interfaces are supported, because the eStream file system is relatively low-functionality (compared to NTFS, for example).

The following file system requests will be supported; the interfaces for them will not be shown here, as they can be found in the DDK documentation.

- Create IRPs, for both new and existing files
- Cleanup and Close IRPs
- Read and Write IRPs:
  - synchronous and asynchronous
  - cached and non-cached
  - paging and non-paging
- Fast I/O reads and writes (with buffers or MDLs)
- File information (get and set) IRPs
- Directory query IRPs
- Volume information (get and set) IRPs
- File system information (get and set) IRPs
- Flush buffer IRPs
- System shutdown IRPs
- Various Fast I/O queries

The EFSD will not handle Directory Notification IRPs, nor will it support hard links (which are supported natively on NTFS on W2000 only): neither of these requests are required, and no expected user functionality will be lost without them. We are presently not supporting byte-locks; this may need to be revisited if the need arises.

In addition to the interfaces to the NT Executive, the EFSD will support various interfaces from other client components; all these will be sent via IOCTL calls. The first ones listed are simple support interfaces; the interfaces between the ECM and the EFSD follow these.

An IOCTL coming in to the kernel—via a DeviceIoControl() call—has the following parameters:

- IOCTL control code
- input buffer pointer
- input buffer size
- output buffer pointer
- output buffer size
- pointer to a 4-byte variable to receive the number of bytes written to the output buffer
- pointer to an OVERLAPPED structure for asynchronous operation (always should be NULL for EFSD)

All of the following interfaces are described in terms of the IOCTL buffers sent and received for each control code.

The following interfaces are called from the controlling client component (StartClient).

Starting and stopping the file system

The eStream FSD will be loaded into the kernel when a system is rebooted; i.e., it is always resident. If applications access files on this FSD via a drive letter, then the file system is implicitly turned off while a symbolic link for that drive letter is not present. Even when a drive letter symlink exists, the EFSD will not accept requests until the START IOCTL is sent.

These IOCTL control codes will be defined for starting and stopping the eStream FSD:

```
IOCTL_EFS_START_FS
IOCTL_EFS_STOP_FS
```

Starting the FSD

The input buffer for the START IOCTL should have the following:

- version id: 4-byte identifier for the client component
- debug flags: 4-byte value indicating the debug level to use

The output buffer for this IOCTL will be filled with the following:

- version id: 4-byte identifier for the EFSD version present
- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BAD_VERSION
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_DUPLICATE_REQUEST
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

A DUPLICATE_REQUEST error is returned if the FSD is already started.

Stopping the FSD

The input buffer for the STOP IOCTL should have the following:

- force: 4-byte value
  - 0: shutdown only if no outstanding files are open
  - 1: shutdown regardless of state of open files

The output buffer for this IOCTL will be filled with the following:

- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_DUPLICATE_REQUEST
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

A DUPLICATE_REQUEST error is returned if the FSD is already stopped.

Cache management interfaces

The following two interfaces are defined for use by the ECM to potentially invalidate data in the NT File Cache.

These IOCTL control codes will be defined for cache management for the eStream FSD:

```
IOCTL_EFS_INVALIDATE_FILE
IOCTL_EFS_INVALIDATE_DIR_CONTENTS
```

Invalidating a file

The input buffer for the INVALIDATE_FILE IOCTL should have the following:

- handle: 4-byte EFSDFileHandle for the open file that must be invalidated

The output buffer for this IOCTL will be filled with the following:

- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_FILE_NOT_OPEN
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

If in fact the file is open, but not present in the NT File Cache, this IOCTL will simply succeed; no error is returned.

Invalidating directory contents

The input buffer for the INVALIDATE_DIR_CONTENTS IOCTL should have the following:

- handle: 4-byte EFSDFileHandle for the open directory whose contents must be invalidated

The output buffer for this IOCTL will be filled with the following:

- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_FILE_NOT_OPEN
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

General file system requests

All file system requests that cannot be completely handled by the EFSD will be passed on to the ECM. Since the ECM is likely to be a user-mode service, the EFSD cannot call it directly; thus these "calls" are made by having the ECM send IOCTLs to the EFSD to get and fulfill requests. Each file system request requires multiple IOCTLs sent from the ECM to the EFSD:

1. The ECM sends an IOCTL to the EFSD to get the next request
2. The ECM sends a second and/or third IOCTL to finish the request

The following IOCTL control codes will be defined by the EFSD for use by the ECM:

```
IOCTL_EFS_GET_REQUEST
IOCTL_EFS_RETRY_REQUEST
IOCTL_EFS_GET_CREATE_NAME
IOCTL_EFS_FINISH_CREATE
IOCTL_EFS_FINISH_CLOSE
IOCTL_EFS_FINISH_READ
IOCTL_EFS_GET_WRITE_DATA
IOCTL_EFS_FINISH_WRITE
IOCTL_EFS_GET_RENAME_TARGET
IOCTL_EFS_FINISH_RENAME
IOCTL_EFS_FINISH_DELETE
IOCTL_EFS_FINISH_METADATA_READ
IOCTL_EFS_FINISH_METADATA_WRITE
```

For the DeviceIoControl() call sending IOCTL_EFS_GET_REQUEST, these parameters are invariant:

- the IO control code will be IOCTL_EFS_GET_REQUEST
- input buffer pointer will be NULL
- input buffer size will be 0
- output buffer must be non-NULL
- output buffer size must be **at least 40 bytes**—this is the largest buffer needed for any request (subject, of course, to slight modifications)
- pointer to bytes returned will be non-NULL
- overlapped pointer will be NULL

The IOCTL_EFS_RETRY_REQUEST is sent by the ECM (or some other user-space client component) to tell the EFSD that, yes, it needs to delete all intermediate information about a request already sent back with a GET_REQUEST call, and put the request back on the list for the ECM to retrieve. This eases implementation issues for the ECM. The input buffer for a RETRY_REQUEST is:

- request id of the previously retrieved request

There is no output buffer for a retry request IOCTL.

What follows is a list of file system requests from the NT I/O Manager, and the IOCTL calls needed from the ECM to service those requests. For all cases, if the EFSD writes to the output buffer for an IOCTL, the "bytes returned" field is written with the number of bytes written.

Create

This is used for both create and open, for files and directories.

GET_REQUEST

The output buffer for the IOCTL_EFS_GET_REQUEST will be filled with the following:

- type: a 4 byte field that indicates a Create request
- request id: a 4 byte field that will be subsequently sent in the calls to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- flags: 4 bytes, one or more of the following ORed together
    - CREATE_ONLY: fail if file exists already
    - OPEN_ONLY: fail if file does not exist already
    - TRUNCATE: overwrite existing file
    - DIRECTORY: create a directory
    - FILE: create a plain file
    - DELETE_ON_CLOSE: delete file on last close
    - IGNORE_CASE: obvious
- permissions: 4 bytes, one or more of the following ORed together
    - READ
    - WRITE
    - EXECUTE
- length of filename: 4 bytes, specifying the byte size needed for the Unicode string sent in the next call

Total size of output buffer: 24 bytes

GET_CREATE_NAME

The input buffer for this IOCTL should have the following data:

- request id: the id sent in the previous call

The output buffer for this IOCTL will be filled with the following information:

- request id for this transaction
- fully qualified name as a counted Unicode string (including drive letter, if any): the length needed was sent back in the GET_REQUEST call

FINISH_CREATE

The input buffer for this call should have the input buffer filled as follows:

- request id: the matching id from the GET_REQUEST call
- status: the NTSTATUS result from this request
- handle: the 4-byte handle for this opened file, that can be used for subsequent file system requests. A unique value will indicate a bad handle, and a failed Create
- a Metadata buffer: the metadata for the created/opened file/directory.

The output buffer for this IOCTL should be NULL.

Note that a TRUNCATE Create request should cause the metadata sent back to reflect the possibly new (zero) length.

Close

This closes a handle of a previously opened file or directory. The EFSD will optionally send the updated metadata for this file in the GET_REQUEST output buffer. If the file has been modified in any way, the metadata fields will be non-zero; else they will all be zero.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Close request
- request id: 4 bytes, for use in subsequent calls for this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes, the handle for the previously opened file
- metadata for this file/directory: 24 bytes
  - creation time stamp
  - modification time stamp
  - file/directory size in bytes
  - attributes (as described above)

Total size of output buffer: 40 bytes

FINISH_CLOSE

The input buffer for this call should contain the following:

- request id for this transaction
- status: the NTSTATUS for this request

Read

This is used for reading file data.

GET_REQUEST

The output buffer for the IOCTL_EFS_GET_REQUEST will be filled with the following:

- type: a 4 byte field that indicates a Read request
- request id: a 4 byte field that will be subsequently sent in the IOCTL to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes, the handle for this previously opened file
- offset: 4 bytes, the file offset, in bytes, to read from
- length: 4 bytes, the length of the read, in bytes

NOTE: The buffer requested in the (offset, length) pair will *not* cross a 4K page boundary.

Total size of output buffer: 24 bytes.

FINISH_READ

The input buffer for this call should have the input buffer filled as follows:

- request id: the matching id from the GET_REQUEST call
- status: the NTSTATUS result from this request
- the number of bytes successfully read; 0 on error
- the data from the read; not present on error

The output buffer for this IOCTL should be NULL.

Write

This is used for writing file data.

GET_REQUEST

The output buffer for this will be filled with the following:

- type: a 4 byte field that indicates a Write request
- request id: a 4 byte field that will be subsequently sent in matching calls for this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes, the handle for this previously opened file
- offset: 4 bytes, the file offset, in bytes, to write to
- length: 4 bytes, the length of the write, in bytes
- file length: 4 bytes, the length the file will be **if** this write succeeds

Total size of output buffer: 28 bytes.

NOTE: The buffer requested in the (offset, length) pair will *not* cross a 4K page boundary.

GET_WRITE_DATA

This IOCTL will have an input buffer with:

- request id for this transaction
- status: if not STATUS_SUCCESS, this ends the request; the output buffer is untouched, and no FINISH_WRITE call is expected.

And the output buffer will be filled with:

- request id
- data buffer for the write—the byte length sent in the previous GET_REQUEST

FINISH_WRITE

For this finishing request IOCTL, the input buffer has these contents:

- request id: the matching id from the GET_REQUEST call
- status: the NTSTATUS result from this request
- bytes actually written; should be equal to requested bytes unless failure occurs.

Rename

This is used for renaming a file or directory.

GET_REQUEST

The output buffer for this IOCTL will be filled with the following:

- type: a 4 byte field that indicates a Rename request
- request id: a 4 byte field that will be subsequently sent in the FINISH_REQUEST call to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory
- length of target name: 4 bytes; the byte length needed for a counted Unicode string for the target name

Total size of output buffer: 20 bytes.

GET_RENAME_TARGET

The input buffer for this call will have the following:

- request id for this transaction
- status: if not STATUS_SUCCESS, then this terminates the request: the output buffer is not touched, and no FINISH_RENAME call should be sent

The output buffer will be filled with the following:

- request id
- target name: a counted Unicode string, using the same number of bytes as sent in the GET_REQUEST output buffer

FINISH_RENAME

The input buffer for this call should have the following:

- request id
- status: NTSTATUS for the transaction

Delete

This is used for deleting a file or directory.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Delete request
- request id: a 4 byte field that will be subsequently sent in the call to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory

Total size of output buffer: 16 bytes.

FINISH_DELETE

The output buffer for this call should be NULL.

The input buffer should have the following contents:

- request id: matching id from the GET_REQUEST call
- status: NTSTATUS of this request

Metadata read

This is used for requesting metadata about a file or directory.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Metadata request
- request id: a 4 byte field that will be subsequently sent in the call to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory

Total size of output buffer: 16 bytes.

FINISH_METADATA_READ

The output buffer for this IOCTL will be NULL.

The input buffer should have the following contents:

- request id: id from the corresponding GET_REQUEST
- status: NTSTATUS for this operation
- the following data about the file or directory:
  - creation time stamp
  - modification time stamp
  - file/directory size in bytes
  - attributes (as described above)

Metadata write

This is used for setting metadata for a file or directory.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Metadata Write request
- request id: a 4 byte field that will be subsequently used for all calls for this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory
- metadata for this file/directory: 24 bytes
  - creation time stamp
  - modification time stamp

- file/directory size in bytes
- attributes (as described above)

Total size of output buffer: 40 bytes.

FINISH_METADATA_WRITE

The input buffer should have the following contents:

- request id: from the previous call
- status: NTSTATUS for this request

Component design

This section is organized in the following manner:

1. General layout of the eStream file system driver
2. General observations about the low level design
3. Organization of data structures
4. Description of the algorithms for communication with the ECM
5. Description of each dispatch routine

Layout

The EFSD will be generally organized in the following manner:

- All major IRPs will have their own dispatch routine.
- All actual I/O requests to the ECM will be generalized from the dispatch routines to a set of routines that handle the communication with the ECM, to isolate this aspect.
- All utility functions will be in their own file or files.

General points

The design of the EFSD will look a lot like the sample FSD from Rajeev Nagar's NT FS Internals book, which looks a whole lot like the Fastfat FSD source from the NT IFS kit.

Here is a list of general points that can be made about the EFSD:

- Any IRP that can be handled asynchronously will be posted to a work queue; this means that the dispatch routine for such an IRP must be able to handle being called in a context other than the original requestor.
- There are no volumes, and no Volume Parameter Block or Volume Control Block. There isn't a VPB for a network redirector; I've verified this with the

LanManager redirector. Hence we don't have to support any operations on a volume in EFS.

- We will not allow the creation of paging files in EFS. There is a bit available for a Create IRP that specifies this, and we can complete the IRP with an unimplemented error return code.
- All file synchronization will be on a File Control Block (FCB) basis, using the standard Resource and PagingIoResource ERESOURCE objects used by the rest of the Windows Executive.
  - User requests will be synchronized by acquiring the main Resource—shared for reads, shared for most writes, and exclusive for file size changes, deletion, etc.
  - Paging I/O requests will be synchronized by acquiring the PagingIoResource—again, shared for reads, shared for most writes. Exclusive access will be needed to set file sizes.
- Most disk file systems have a resource associated with a VCB, which is acquired exclusively for creation/deletion etc. We will have a global EFS resource for this, since there are no VCBs.
- Asynchronous requests will be handled by posting the IRP to the CriticalWorkQueue, and marking the IRP as pending.
  - A common worker routine will be used for all async posts, which will dispatch the IRP to the appropriate real IRP routine when it's invoked.
  - An async request will be defined as one that IoIsOperationSynchronous() returns false, and the EFSD is the top-level component (see below)
- The EFSD will track the top-level IRP for the thread whose context it is running in. In particular,
  - No async processing request will be honored unless the EFSD is the top-level component
  - No cache manager requests will be made unless the EFSD is the top-level component
  - EndOfFile size—that is, the true size of the file—will not be extended or changed by paging I/O
- EFS will not support holes in files, and hence the ValidDataLength FCB field will be set to disable this. This means the AllocationSize for an eStream file/directory will always be equal to the EOF size.
- Most fast I/O routines will be supported in EFS. We will use the FSRTL supplied routines for fast reads and writes.
- All cache manager resource acquire/release callbacks will be supported. All will point to common routines that simply acquire or release the main or paging I/O resource for the FCB. The Context pointer passed into all of them will be the FCB for the stream.
- Synchronous read/write requests will update the CurrentByteOffset in the File object.
- Each Create will result in a unique Context Control Block (CCB) data structure; this will be small, and only hold those few fields needed:
  - For the Directory Control IRP, a CCB needs to hold the current entry index and the pattern originally used—for subsequent queries

  - A field for various flags
- A single FCB will represent all current open instances of a file. When a Create request comes in, the EFSD will search the current open FCBs to try to find one matching this file/directory name.
  - For now, this will be a hash table on the file name. We can improve this as needed.
  - The EFS global resource must be acquired exclusively:
    - before the global FCB data structure is searched. **Why? If it's just a read, can't we acquire it non-exclusively?**
    - before a new FCB is added to the list
    - before an FCB is deleted from the list
- EFS will not support open by file ID; hence the FileInternalInformation class for a File Information IRP will not be supported.
- Actual I/O will be directed to standard routines in a separate file, so they can be isolated and updated easily as our method of transferring data changes.
- Here's how to do file/directory renames:
  - The I/O manager will send to the EFSD this sequence:
    - Create for source
    - Create for target, with the SL_OPEN_TARGET_DIRECTORY flag set
    - Set Information with a Rename request for the source, sending the target directory FileObject handle and the target name in the FileInformationClass record.
  - EFSD needs to do this:
    - When it receives the Create for the target and the target *directory* exists, return STATUS_SUCCESS, and change the name in the FileObject to the basename of the target (the full pathname of the target is sent in), and set the Status.Information to FILE EXISTS or FILE DOES NOT EXIST, as appropriate. If the target directory doesn't even exist, return PATH NOT FOUND.
    - When it receives the Set Info request, if all the flags check out (e.g., if the file exists, ReplaceExisting must be TRUE), send a Re-name request to the ECM.
- Reads and writes to only regular files will be supported, not to directories.
- Any code that touches user buffers or can call routines that may throw exceptions must be guarded by a try/except block.
- Some tips on memory allocation (from /perforce-doc-dir/osrdocs/defensive-driv.html)
  - We will use our own memory allocation/deallocation routines, instead of ExAllocatePool() et al. directly
  - These routines can do various checks for trashing memory:
    - fill allocated memory with a pre-defined bit pattern, instead of ze-roes; fill deallocated memory with a different pattern.
    - allocate a header/trailer with standard information, like where allo-cated, from what pool, etc.

    - change the bit pattern in the header/trailer on deallocation, and look for freeing memory twice
- We probably want to allocate using lookaside lists, since we'll be allocating and deallocating smallish chunks of memory for our data structures.

Data structures

The following are major data structures used internally by the EFSD. Data structures used to communicate with the ECM are described in the following section.

NodeIdentifier

A NodeIdentifier is a simple structure that starts all other structures used in the EFSD. This makes a good debugging check to insure that we receive and are operating on the right type of data. It consists of two fields: an identifier field, and the total structure size.

FCB

The FCB is a critical data structure for the file system driver. There is one FCB structure allocated for each unique file or directory that is currently open—regardless of how many open handles there are for this entry. Multiple CCB structures can point to a single FCB.

Logically at least, part of an FCB is exposed to the Cache Manager and VMM to support caching and paging I/O. We will follow the example of Rajeev Nagar's book, and embed the FSRTL_COMMON_FCB_HEADER and other required structures directly in an FCB.

Here are the basic contents of an EFSD FCB:

- The required FCB contents above
- An open handle count: incremented on Create, decremented on Cleanup
- A reference count: incremented on Create, decremented on Close. The semantics of NT file requests require these two be used together to determine when an FCB can be deleted
- A pointer to the next FCB on its hash list
- A pointer to the first CCB opened for this FCB
- The fully qualified name of the file/directory
- The Metadata associated with this file/directory
- A SHARE_ACCESS structure used to check sharing violations
- Various flag bits

CCB

A CCB represents a currently opened handle to a file or directory. If two processes have the same file opened, there will be a unique CCB allocated for each process.

Here are the basic contents of an EFSD CCB:

- A pointer to its corresponding FCB
- A pointer to the next CCB opened for its FCB
- A pointer to the FileObject opened for this file/directory
- The current file index for a directory query
- The directory search pattern used for directory queries for this opened handle
- Various flag bits needed

IrpContext

An IrpContext structure encapsulates the interesting data from an IRP, and the current stack location, for easy access. This structure is allocated on entry to dispatch routines, and used during processing, before being deallocated on exit.

Communicating with the ECM

I tend to divide a file system driver into two logical parts:

1. A front-end that understands the NT FSD interfaces and semantics
2. A back-end that actually perform the requested actions

Our back-end is the (admittedly ugly) interface for communicating with the ECM, which currently sits in user-space. It's very important to design the ECM such that its front-end and back-end are nicely separated: since the ECM may move to kernel space, or we might find better interfaces for them to communicate with each other, we need to design with this in mind.

Given the current ECM interfaces defined above, here is a basic design to handle them:

- An EfsdRequest object will be created for each request that must be handled by the ECM.
- Each dispatch routine that results in a request to the ECM will allocate an EfsdRequest and send it to a common routine for further processing.
- New requests will be placed in a NewRequest queue.
- Requests that have been "sent" to the ECM, but not yet finished, will be removed from the NewRequest queue and placed in a PendingRequest list.
- Finishing a request entails removing it from the PendingRequest list, returning the contents to the dispatch routine, and destroying the request object.

Data structures

EfsdRequest

This contains:

- request id: a number that uniquely identifies this request
- type: the type of request (e.g., Create, Write)
- FCB pointer for the file/directory for this request
- IrpContext pointer for this request
- a kernel event object used for signaling that the request is satisfied

NewRequestSemaphore

The EFSD device object's device extension will contain a semaphore dispatcher object, initialized to non-signaled, and with an initial limit of MAX_LONG. When a request is added to the NewRequest queue, this semaphore is released (and the count is incremented by 1); the GET_REQUEST IOCTL will wait on this semaphore object (which decrements the count by 1).

NewRequestQueue

This is actually a kernel-managed interlocked list that is allocated in the device extension area, guarded by a spin lock that's also located in the device extension. Requests will be added to the tail, retrieved from the head.

PendingRequestList

This list must be searched when an ECM non-GET_REQUEST IOCTL is received, so we can't use an interlocked list. We'll use a global single-linked list structure, with elements allocated from a dedicated lookaside list using non-paged memory. Before a thread can access the list elements, it must acquire a mutex.

Algorithm

- All dispatch routines, and asynchronous read/write routines, will call LowLevel-PostRequest() to have their requests satisfied. LowLevelPostRequest() is itself synchronous; that is, the code calling it is something like this:

```
status = LowLevelPostRequest(fcb, irp_context);
     // we're done;
     // do all deallocation and cleanup needed
IoCompleteRequest(irp, ...);
```

- LowLevelPostRequest() will do the following:

```
if this is a read or write IrpContext
    see how many requests are needed to satisfy the IRP: can't span page boundary
    allocate the N requests
    allocate an array of N event pointers to hold the request events
    assign the pointers to the array
    if > THREAD_WAIT_OBJECTS
        allocate an array of N PKWAIT_BLOCKS
```

```
else if this is a directory query IrpContext
    look at the FileSize of the directory FCB
    allocate enough read requests to read all directory data from the ECM
    as above, allocate an array of event pointers, wait objects (if necessary)
else
    allocate a new EfsdRequest for the incoming FCB and IrpContext
place all requests on the NewRequestQueue, using ExInterlockedInsertTailList()
call KeReleaseSemaphore() on the NewRequestSemaphore
if multiple requests generated
    call KeWaitForMultipleObjects() on the request object's event
else
    do a KeWaitForSingleObject() on the request object's event
when the event(s) is/are signaled
    fill in the values into the IRP
    deallocate the EfsdRequest(s)
    return status
```

- When a GET_REQUEST IOCTL comes in from the ECM, the EFSD will do the following:

```
do a KeWaitForSingleObject() on the NewRequestSemaphore
remove the first request from the NewRequestQueue,
                    using ExInterlockedRemoveHeadList()
lock the PendingRequestList mutex
place this request on this list
release the mutex
fill in the IOCTL output buffer identifying the request
complete the IOCTL IRP
```

- When a RETRY_REQUEST IOCTL is received, the following takes place:

```
lock the PendingRequestList mutex
search the list by request id; error if not found
remove the request from the list
release the mutex
enqueue the request on the NewRequestQueue—on the list head
release the NewRequestSemaphore
complete the IOCTL IRP
```

- When any "finishing" IOCTL is received—i.e., the second of two or the third of three—the following is done:

```
acquire the PendingRequestList mutex
search the list by request id; error if not found
remove the request from the list
release the mutex
do all buffer copying, set all flags,
            and otherwise insure the input IRP has the correct state
signal the EfsdRequest event
complete the IOCTL IRP
```

- When any other request IOCTL is received—e.g., the second of three—this is done:

```
acquire the PendingRequestList mutex
search the list by request id; error if not found
release the mutex
fill in the output buffer of the IOCTL as appropriate
complete the IOCTL IRP
```

Dispatch routines

DriverEntry

This does a whole slew of initialization, including the dispatch table, fast I/O table, the cache callbacks, the FCB hash table and its synchronization object, creates the FS device object, and sets up the interface with the ECM.

Create

There is one Create routine; there will be no async processing of Create requests. Ultimately, its job is to send a create request to the ECM, and return SUCCESS or not to its caller. Here is a general algorithm for this routine.

```
create an IrpContext
if a page file is requested
    return error
generate the absolute pathname of the requested file:
    if OPEN_TARGET_DIRECTORY specified
        if OPEN_ONLY not specified
            return error
        generate the pathname of the parent directory of the requested file
    if a related file object is specified
        if the related file is not a directory
            return error
        if the related filename doesn't start with '/'
            return error
        if the input filename starts with '/'
            return error
        concatenate the input filename with the related file directory
    else use the input filename
        if the input filename does not start with '/'
            return error
acquire the global EFS resource exclusively
search the FCB hash table for this file by name
if not found
    call LowLevelPostRequest() to send the request to the ECM
    if error is returned from ECM
        return the correct error: FILE NOT FOUND or PATH NOT FOUND
    create a new FCB and add to the hash table
    call IoSetShareAccess() for this FCB
else
    check input attributes/flags against those in FCB
        if opening for write or delete on close
            call MmFlushImageSection()
            if this fails
```

```
            return error
        if failing mismatch—e.g., if IoCheckShareAccess() fails
            return error
    create a new CCB for this file
    set all appropriate flags on CCB and/or FCB
        COMMON_FCB_HEADER flag is set to FastIoIsPossible
    set all fields on input FileObject:
        write through flag
        FsContext points to common FCB header
        FsContext2 points to CCB
    if OPEN_TARGET_DIRECTORY is specified
        search for the input target object:
            look for this in the FCB hash table
            if not found
                send a Create request for this file to the ECM
            if this target filename exists
            set Status.Information to FILE_EXISTS
        else
            set Status.Information to FILE_DOES_NOT_EXIST
        if a Create was sent to the ECM for the input FileObject
            send a Close request for the input target to the ECM
        change the name in the FileObject to the basename of this file
        the CCB and FCB remain opened for the target directory
    all necessary data structures should be deallocated, for success and error
    release the global EFS resource
```

Cleanup

No async posting of Cleanup requests will be done. Algorithm:

```
    acquire the global EFS resource, and the FCB Resource, for this file, exclusively
    if this file is marked for deletion
        if this is the last open handle for the file
            acquire the PagingIoResource exclusively
            set the file size in the FCB to 0
            release the PagingIoResource
            purge the cache, if necessary, with MmFlushImageSection()
            call LowLevelPostRequest() to send a Delete request to the ECM
    decrement the count of open handles in the FCB
    if caching is on
        flush the cache by calling CcUnitializeCacheMaps()
    any time stamps must be updated if accesses were done using fast I/O
    set the FO_CLEANUP_COMPLETE flag in the FileObject
    call IoRemoveShareAccess()
    release the global EFS and FCB Resources
```

Close

There will be no async posting of Close requests. Note that we only send a Close request to the ECM if this is the last close for an open file—i.e., we're matching Close requests with Create requests.

Algorithm:

```
acquire the global EFS resource exclusively and the FCB Resource
deallocate the CCB
decrement the reference count for the FCB
if the FCB ref count is now 0
    remove the FCB from the hash table
    deallocate the FCB
    call LowLevelPostRequest() to send a Close request to the ECM,
                                        updating metadata if necessary
release the global EFS resource and the FCB Resource
```

Read

Reads will definitely be open to async posting. A general algorithm:

```
if the read length is 0
    return success
if the target file object is a directory
    return error

if this IRP can be handled async
    post for async processing

if this read is non-buffered, and it's not for paging I/O, and
                    there is a mapped data section object for this file
    acquire the FCB main Resource exclusive, and the PagingIoResource shared
    call CcCacheFlush() on the range of this read (current byte offset, length)
    release the FCB resources
if this is for paging I/O
    acquire the PagingIoResource shared
else
    acquire the main Resource shared
if this read starts beyond EOF
    return EOF
if the read length goes beyond EOF
    truncate the length
if this is a buffered read
    if the PrivateCacheMap is NULL
        call CcInitializeCacheMap()
    if this is an MDL read
        call CcReadMdl()
    else
        call CcCopyRead(), using either an allocated MDL or the UserBuffer

else (it's non-buffered)
    call LowLevelPostRequest() to send a Read request to the ECM

if this is a synchronous, non-paging read
    update the current byte offset in the FileObject
set the number of bytes read in the Status.Information field
release any acquired resource and deallocate appropriate data structures
```

Write

Writes will definitely be open to async posting. A general algorithm:

```
if the write length is 0
    return success
if the input file is a directory
    return error
if the file not opened with write permissions
    return error

if this IRP can be processed asynchronously
    post for async processing

if this is a buffered write
    call CcCanIWrite()
    if false
        we have a hard error; fail
if this is paging I/O
    acquire the PagingIoResource shared
else
    acquire the main Resource shared
if the length is
        (Low == FILE_WRITE_TO_END_OF_FILE) && (High == 0xffffffff)
    we're to write at EOF
if this is a non-paging, non-buffered write, and
                            there is a mapped data section object for this file
    acquire the global EFS resource exclusive
    acquire the PagingIoResource shared, starving exclusive waiters
    call CcCacheFlush() on the range for this write
    release the PagingIoResource
    return error if the cache flush failed
    acquire and release the PagingIoResource exclusive (to serialize)
    call CcPurgeCacheSection() on the range for this write
    release the global EFS resource

if this is a paging I/O write
    if the starting offset is beyond the current EOF
        return success
    if the ending offset is beyond the current EOF
        truncate the write length to EOF

if this is a buffered write
    if the private cache map is NULL
        call CcInitializeCacheMap() for this file
    if the write will extend the file size
        release the resource acquired
        re-acquire the resource exclusive
        call CcSetFileSizes() inform the cache manager
    if this is an MDL write
        call CcPrepareMdlWrite()
    else
        call CcCopyWrite(), using either an allocated MDL or the UserBuffer

else
    call LowLevelPostRequest() to send the write request to the ECM

set the number of bytes written in the Status.Information field
update the file size fields in the FCB if this write extends the length
if this is a non-paging write
```

```
        set the CCB modification flag
        if this write is synchronous
            update the CurrentByteOffset field in the FileObject
    release any acquired resource and deallocate appropriate data structures
```

Fast I/O Read

Initially at least, we'll just set the fast I/O read routine to FsRtlCopyRead().

Fast I/O Write

Initially at least, we'll just set the fast I/O write routine to FsRtlCopyWrite().

Fast I/O Query Basic Info

This will just fill in the basic info buffer with the data in the FCB.

Fast I/O Query Standard Info

This will just fill in the standard info buffer with the data in the FCB.

Fast I/O Query Open

This will just fill in the network open info buffer with the data in the FCB, if the file exists. Some empirical observations I've made using NTFS:

- Regardless of whether the file exists or not, this will return TRUE (all fast I/O routines are boolean)
- If the file does not exist, it will set the EOF size in the buffer to 0. The AllocationSize must be non-zero. All other fields seem to be don't cares.
- If the file exists but is zero length, then both EOF and AllocationSize will be 0.
- The IRP sent to this routine is for an IRP_MJ_CREATE; we can use more than just the name to identify the file, but also the security characteristics or whatever else is sent in the IRP.

File Query Info

Standard queries will be supported; these however will not:

- FileInternalInformation—no OPEN_BY_FILE_ID
- FileEaInformation—no EA data
- FileCompressionInformation—no on-disk compression
- FileStreamInformation—no multiple streams

File Set Info

These actions will be supported:

- EndOfFile size changes
- AllocationSize changes
- Time stamp changes
- File position changes
- File disposition changes—delete pending
- File rename requests

Here is a general algorithm:

```
if AllocationSize or EOF size change
    if the new size is smaller than the current size
        call MmCanFileBeTruncated(), to ask the VMM if this is okay
        if yes
            call CcSetFileSizes()
        else
            return STATUS_USER_MAPPED_FILE error
    else
        send a Metadata Set request to the ECM with new, extended size
        if status returned is error
            return error (disk space full)
    update AllocationSize and FileSize fields of required FCB header

else if time stamp change
    send Metadata Set request to ECM
    if error returned
        return with error

else if file position change
    update FileObject CurrentByteOffset field

else if disposition (delete) change
    if the Delete flag in the IRP not set
        clear delete on close FCB flag
    if file already marked for delete on close
        return success
    if file not open with write permission
        return error
    if MmFlushImageSection() fails
        return error
    if this is a directory, and the directory is not empty
        return error
    set delete on close flag in FCB

else if rename change
    if the source name is for a directory
        if the directory has any open files/directories under it
            return error

    if Parameters.SetFile.FileObject is NULL (simple rename)
        target dirname is the same as dirname of IRP FileObject
        target basename is Buffer.FileName

    else
        if Buffer.RootDirectory is NULL (fully-qualified rename)
```

```
            fully qualified target name is Buffer.FileName

        else  (relative rename)
            call ObReferenceObjectByHandle() to get the file object of the relative directory
            from file object, get (fully qualified) dirname of target
            append Buffer.Filename to root directory dirname to get fully qualified target

    if the target name isn't on EFS
        return error
    if target exists
        if Parameters.SetFile.ReplaceIfExists is FALSE
            return error
        if target is a directory
            return error
        if target is read-only
            return error
        if target has any open handles to it
            return error

    call LowLevelPostRequest() to send a Rename request to the ECM
```

Directory Query

Directory queries turn into directory read requests to the ECM. The EFSD will take the contents of the read buffers and fill the IRP_MN_QUERY_DIRECTORY buffers sent to it from the NT Executive by parsing the directory contents.

Note: this design does not use the NT Cache Manager for metadata or for directory entries, nor does the EFSD store the directory contents anywhere in its data structures. It will always go back to the ECM for directory queries. Given that most directory queries occur in this order:

Create
Directory query
Close

unless we can cache the contents in a location more persistent than an FCB, we will need to resubmit the request to the ECM for each new directory query. If this poses a performance problem, we need to handle it then.

Directory queries are subject to async processing.

This is an ugly NT interface. Here are some general points regarding directory queries, and how they will be implemented on EFS:

- These requests come in from the I/O Manager in a context-sensitive sequence. I.e., a request will come for the initial N directory entries; the next request will be for the next M entries; etc. Kind of like strtok().
- Thus, state must be maintained from request to request. This state will be kept in the CCB for a file stream, and consists of:

    - Pattern sent in on first request
    - Index of n'th entry to start retrieving with
- My experience is that the INDEX_SPECIFIED flag is **never** set in a directory control query, even on queries subsequent to the initial one.
- For EFSD, the FileIndex will represent the offset of the fixed-length portion of a directory entry as returned by the ECM.
- Initially, at least, **all** directory queries will cause the EFSD to read all the entries for that directory from the ECM. We can modify this later if needed.

Here is a general algorithm, based on the sources for the Fastfat driver:

```
if the FileObject is not for a directory
    return error

post this for async handling, to read all directory pages from the ECM

if the CCB pattern field is empty and the CCB flag "match all" isn't set
    this is the initial query
    acquire the FCB Resource exclusive
else
    acquire the FCB Resource shared

get a pointer to the input buffer, using either an MDL or the UserBuffer
if this is the initial query
    parse the FileName pattern
    save the pattern in the CCB
    if the pattern is "*"
        set the "match all" flag in the CCB

if SL_RESTART_SCAN is specified
    use an index of 0 for the query
else if SL_INDEX_SPECIFIED is specified
    use the input index for the query
else
    if this is the initial query
        use an index of 0
    else
        use the index saved in the CCB

start with the directory entry corresponding to the starting index
if this index is beyond the number of entries in the directory, and this is not the initial query
    return STATUS_NO_MORE_FILES
while there are more directory entries
    if the directory entry name matches the pattern in the CCB,
                                        using FsRtlIsNameInExpression()
        if there isn't room in the buffer for this entry
            break
        write the entry in the input buffer
        if there is a next directory entry beyond the current one
            the FileIndex field is set to the offset of this next directory entry
        else
            the FileIndex field is set to 0
        if there is a previously written entry
            fix up the NextEntryOffset in the previous entry to the byte offset of this entry
```

```
        if SL_RETURN_SINGLE_ENTRY specified
            break
        8-byte align the current pointer in the user buffer
    advance to next directory entry

if we wrote nothing
    if we stopped because of no room
        return STATUS_BUFFER_OVERFLOW
    else
        return STATUS_NO_SUCH_FILE

update the index field in the CCB
Status.Information is set to the number of bytes written
return STATUS_SUCCESS
```

File System Query Info

Empirically, I've noticed that the LanMan redirector returns failure for most of these requests. So, except for any user-defined FSCTL requests we want to define, I'm going to fail all of these until it turns out we need to do otherwise.

File System Set Info

Ditto for this IRP type too.

Volume Query Info

We at least need to minimally implement these requests:

- FileFsAttributeInformation
- FileFsVolumeInformation
- FileFsDeviceInformation

This will be handled solely by the EFSD; the request will not go out to the ECM. File system size requests will not be handled.

Volume Set Info

We will fail all requests of this type.

Flush Buffers

A buffer flush request for a file stream will mean the following:

- If the file stream isn't buffered, return immediately
- The FCB main Resource is acquired exclusive
- The Cache Manager is told to flush the buffer for the byte range of the file
- The resource is released

A buffer flush for a directory is a successful NOP.

The buffer flush request will insure that contents in the NT File Cache are written to the ECM (as a normal write request); the buffer flush request itself will not be propagated to the ECM.

Testing design

Unit testing plans

The EFSD will be tested apart from integrating with the ECM or the rest of the eStream client. Some points:

- There will be a (relatively) simple stand-in user process for the ECM, to get requests from the EFSD and handle them locally.
- As much EFSD functionality as possible will be done using user-mode test cases (e.g., open files, read/write files, delete files, etc.).
- Some functionality may need to be unit tested using another kernel-mode driver to send explicit IRPs to the EFSD.
- Filemon will be used to monitor the requests being handled by the EFSD.

Stress testing plans

I've heard of a file system filter driver test package available from Microsoft. This is probably the best way we have of stress testing the EFSD.

Coverage testing plans

We'll try to measure coverage on the EFSD. If there is a general kernel-mode method for measuring coverage that's used company-wide, we'll exploit it. Otherwise, there will be some primitive self-coverage instrumentation conditionally inserted in the driver code that we can use at least for major code paths.

Cross-component testing plans

There's not much to do here apart from normal interactions with the ECM.

Upgrading/Supportability/Deployment design

For supportability, there will be solid debugging aids—using printfs—built into the EFSD sources. Additionally, aside from good error codes returned from its interfaces, the EFSD will explore diagnostic traces optionally dumped for deployment.

Issues with stakes in the ground

- At present, I am assuming that there is a single drive letter associated with the EFSD, though there's no technical reason why this must be so. If indeed we organize the client system and the eStream file hierarchies to have multiple drive letters, either the EFSD or the ECM will need to parse the drive letter and do the right thing.
- This design assumes that 8.3 DOS-style filenames need not be supported. Adding such support will increase the complexity of the EFSD, as well as many other components in the eStream system: on the client, on various servers, and on the content builder.
- No support is provided in this design for:
    a. byte locks
    b. directory notification
    c. file open by file id
    d. file system control requests

Open Issues

1. I'm unclear on how to use the NT File Cache for metadata and directory contents. For now, I'm ignoring such matters, and we will only be caching file contents.
2. I do not know how to hook up the EFSD to UNC names. That is, I don't know how to set things up to have all file accesses like \\ASP1\Office\winword.exe be directed to the EFSD by the NT I/O Manager.
3. This design doesn't cover exactly how IRPs are posted for asynchronous processing. The SFSD example in Rajeev Nagar's book really isn't sufficient for some of what we need to do. Also, it's unclear to me what value there is to returning STATUS_PENDING and handling a request asynchronously if the caller is blocking anyway.

EStream Client Functionality:

- ⇨ Installation of eStream client code
  - o Use browser to contact ASP Web Server, download bits to be installed.
  - o Install z: file system hooks & setup to have z: mounted at system boot.
  - o Install eStream client code, which services z: file sys requests from local cache or from servers & which handles sideband communication w/ servers, and setup to activate estream client code at system boot.
  - o Install NoCluster.sys to disable page fault clustering at system boot.
  - o Install eStream browser plug-in, which can receive messages from ASP Per-User Account Server alerting eStream client when new app purchased. [Sending unsolicited messages may not be possible thru firewall.]
- ⇨ Execution of eStream client code
  - o Respond to z: file sys requests. For apps w/ active online connection(s), user sees the detailed contents on z: that one would see if one had installed the apps locally, though copy access may be controlled. For apps to which the user has obtained offline access, user also sees the appropriate detailed contents on z: (although the files are actually in eStream client-managed memory on local disk). For each app whose connection is currently inactive, user sees a placeholder file entry on the z: file system (on which the user can double-click to launch an active connection).
  - o Establish/terminate session logins to ASP Per-User Account Server, upon user request or upon receiving app purchase message from browser plugin.
  - o Obtain/cache unique certificates for purchased applications from ASP Per-User Account Server.
- ⇨ Register with ASP
  - o Use browser to contact ASP Web Server.
  - o Follow ASP process to register.
  - o User obtains login/password, used by estream client code for sessions.
  - o ASP records user's login/password on ASP Per-User Account Server.
- ⇨ Purchase of application
  - o Use browser to contact ASP Web Server.
  - o Follow ASP process to buy app; user is given unique certificate for app.
  - o App purchase & certificate recorded on ASP Per-User Account Server.
  - o User is directed to go to client & request app installation and/or ASP Per-User Account Server attempts to send message to eStream browser plug-in on user's preferred client system (if any), so client can begin app install.
- ⇨ Installation of application
  - o Send unique certificate for application to appropriate ASP DRM Server, get back id for closest/best App Server & a session id.
  - o Contact designated App Server using id info, download meta-data about app, potentially including registry/DLL/filesys spoofing info, prefetching info, initial cache contents for app. For offline installation, obtain all files.
  - o Perform initial installation & setup for app, after checking system for previously installed version of app & issuing any appropriate warnings.

⇨ Execution of application
  - Send unique certificate for application to appropriate ASP DRM Server, get back id for closest/best App Server & session id.
  - Contact designated App Server using id info, request file system data as necessary. Respond to running application's requests, collect usage data. Cache portions of application, file system info, & user preference info.
  - Detect server connection issues (apparent loss of connection or connection response below acceptable threshold); negotiate with ASP DRM Server for alternative connection if need be.
  - At exit from application (or at other selected times), save portions of cache to client nonvolatile memory. Upload usage information to ASP Per-User Account Server.

⇨ Uninstallation of application
  - Remove all registry/DLL/filesys changes associated with app installation.
  - Remove all meta-data about app.

⇨ Uninstallation of eStream client code
  - Remove z: file system hooks, eStream client code, & nocluster.sys.

EStream Server Functionality in terms of kinds of eStream Servers responding to Clients [may be embodied in any number of physical server computer systems]:

1. App server
   - functionally read-only
   - serves .exes, .dlls, etc.
   - contains install info (aka, eStream sets)
2. ASP web server
   - used to get eStream client bits
   - eStream browser plug-in
   - handle other user queries, e.g., concerning available apps, current billing status
3. Per-user account server
   - registration info, issue serial numbers for purchased apps
   - accept/store uploaded info about app usage
   - perhaps: user preferences for each app
4. DRM server
   - authentification of users
   - validate app licenses, track outstanding offline licenses
   - hands out licenses for #1 above

Estream Server Management/Maintenance Functionality

⇨ Install/maintain eStream apps [aka Builder]
  - Provide tool/methods to generate initial meta-data about app, including registry/DLL/file spoofing info, initial prefetching info, initial cache contents, etc.
  - Provide tool/methods to place app & meta-data into public access area and to remove from public access areas
  - Update meta-data as appropriate to reflect uploaded client usage info

⇨ Handle server traffic
  - Support trouble-shooting of performance or correctness problems

  - Perform automated load balancing
  - Support online addition/reconfiguration of servers
- ⇨ Provide tools to process uploaded app usage info.

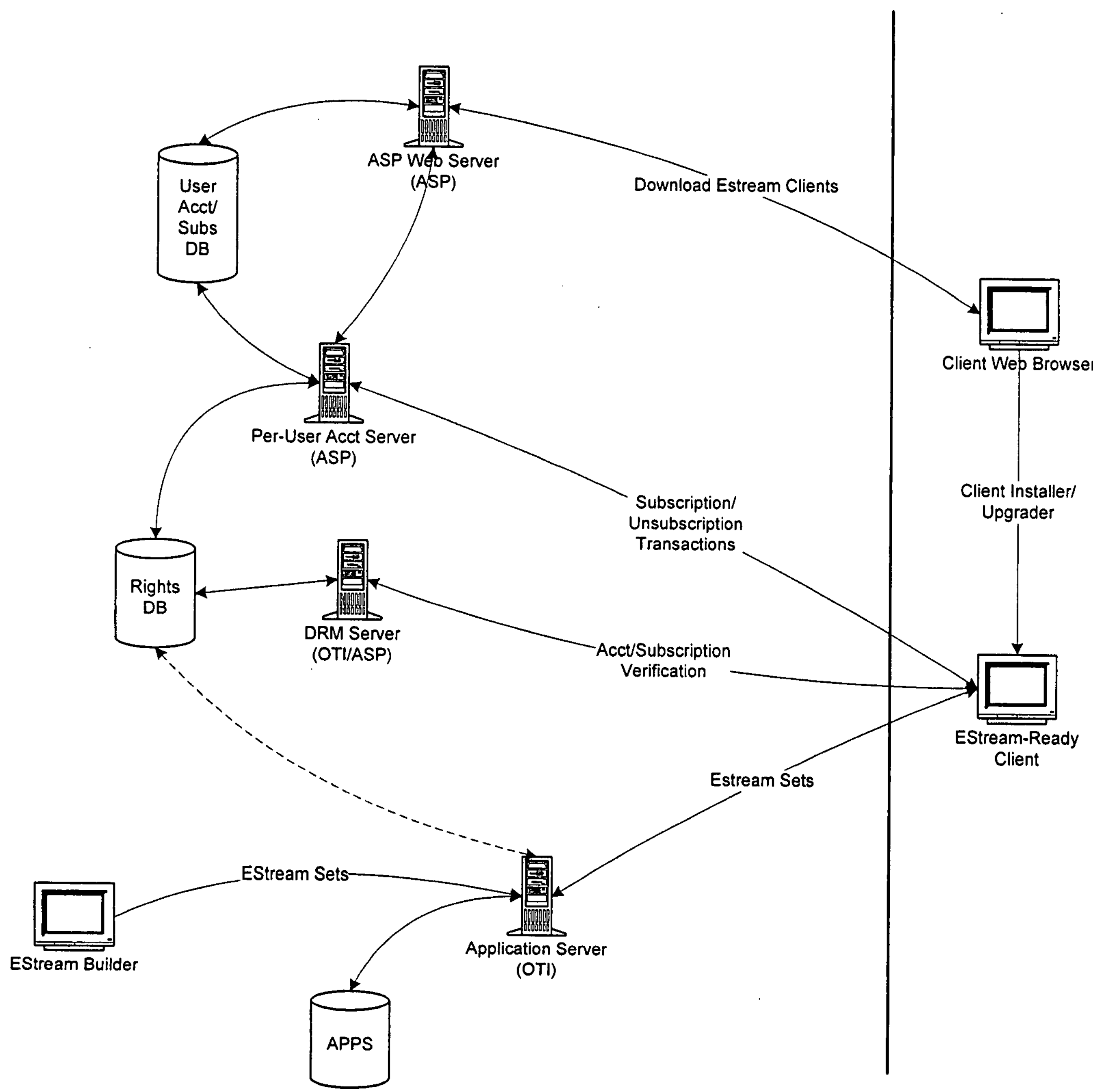
Open functionality questions:

- ⇨ Supporting time-based charge for app-usage (e.g., rent by minutes of usage) complicates the design & may engender customer support/satisfaction issues. Do ASPs want/need this support? [Prefer to steer them away from this model.]
- ⇨ How should we handle minor upgrades/patches of apps (i.e., service packs)? One method to allow active use of previous versions plus availability of new versions without treating new versions as if they were entirely new applications would be file versioning.

EStream 1.0 Top Level Component Breakdown * Revision 0.1 * [redacted]
Client system components

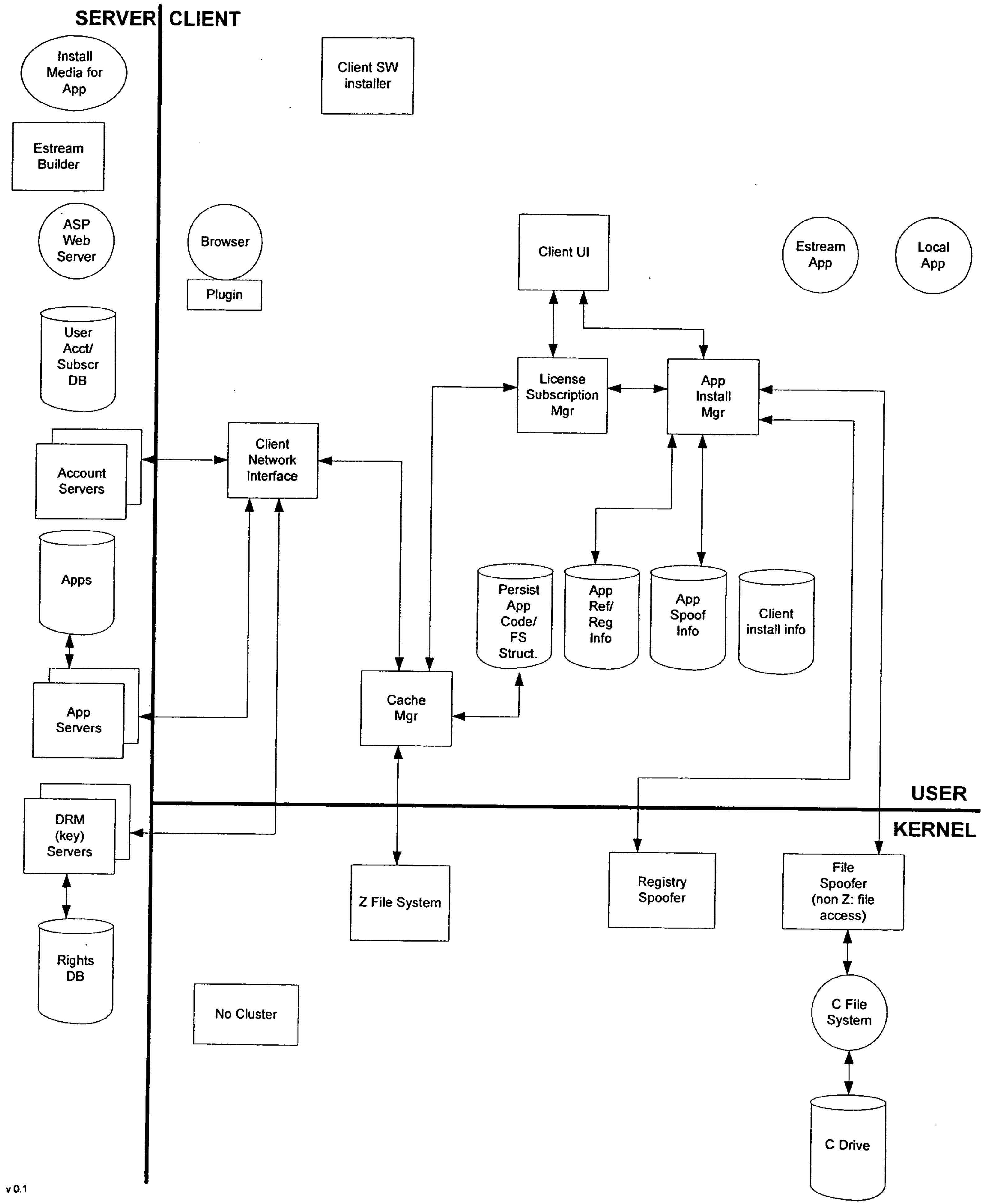
- Z: File system manager [1]
  - Handles all z: file system requests generated on client
  - Makes requests to EStream cache manager
  - Attempts to filter references that suggest software piracy activity
- EStream client core
  - Session manager [12]
    - Handles establishing/terminating ASP sessions
    - Negotiates for app license & security using user unique certificate
    - Invoked either by eStream client user interface or by cache mgr
  - Cache manager [2]
    - Responds to Z: file system manager requests
    - Maintains client cache of app & file system data/metadata
    - Requests info as necessary from Estream client networking
    - Requests session/license for non-mounted apps from session mgr
    - Consumes/gathers profiling/feedback data
  - File manager [3]
    - Provides interface to all eStream created/maintained client files
    - Gets requests from cache mgr, session mgr, file mgr/spoofer, registry mgr/spoofer, app install/deinstall, client install/deinstall
- Estream client network interface [8]
  - Handles requests from EStream cache manager
  - Handles protocol interface to/from server
  - Performs compression/decompression, encryption/decryption of packets
  - Detects network problems & reports to session manager for renegotiation
- EStream client user interface [5]
  - Displays error/info messages from any part of eStream code to user
  - Solicits/obtains info (e.g., login/password, app license) from user
- EStream file system manager/spoofer [6]
  - Filters all non-z: file sys requests, redirects non-z: file refs as appropriate
  - Supports operation of eStreamed apps
  - Avoids eStreamed apps interfering with non-eStreamed apps
- Estream registry manager/spoofer [7]
  - Filters all registry refs, handles registry contents for/about eStreamed apps
- EStream application installer/deinstaller [14]
  - Obtains app spoofing/registry/prefs info & initial cache/profile data
  - Prepares system to be able to run app on user request
  - Supports deinstallation of app
- EStream client code installer/deinstaller [13]
  - Installs all client Estream code components
  - Supports deinstallation of all eStream components
- NoCluster.sys [4]
  - Disables page fault clustering in the kernel
- Estream browser plugin
  - Optional EStream component which fields unsolicited server messages

eStream Client-Server Diagram



v 0.1

eStream High-Level Design Diagram



eStream File System Straw Man Proposal

Version 0.5

[redacted]

Purpose

The purpose of this document is to present a concrete proposal for the functioning of the eStream file system. In many places, I make some sweeping generalizations about how things should work without describing the data structures and interfaces involved in implementing them. This document should eventually involve into a design specification.

Issues Not Covered

This document does not attempt to cover all issues present in designing the eStream 1.0 product. In particular, the overall authentication/licensing/security architecture is not covered in detail here. It is expected that the security functionality will be mostly orthogonal to the design of the basic file system functionality.

Background

There are a number of different networked file systems out there. Many of them share some requirements with eStream. For example, AFS performs client-side on-disk caching, while Coda handles serious server redundancy and disconnected operation. Personally, I believe that AFS and Coda are the file systems whose designs are most relevant to us. For those interested in further background reading, you might also want to look at papers covering NFS, CIFS, xFS, DFS, and Zebra.

Single File System Name Space

Many modern distributed file systems present the network file system as a single tree mounted at some location on the client system, regardless of which server hosts the data. (In fact, with AFS, every file on every server in the world can be accessed through a path starting with /afs on the client, assuming the client can reach that server and has sufficient privileges to do so.) Compared with systems like NFS and Windows sharing, where each share is mounted in a different location on the client, the single name space provides greater ease of use.

The eStream file system would present one universal logical file system. Regardless of which ASP provider supplies a particular volume, that volume will always be referenced via the same path on the eStream file system. That this is desirable or even feasible is predicated on the assumption that OTI is the only entity providing all eStream sets. Each volume must get a unique identifier and a unique location to be mounted in the file system hierarchy. If two different ASPs privide the same volume ID, then the contents of those volumes must be identical. This way, we don't have to tag things in the cache based on what ASP they came from, and the cache manager doesn't need to know anything about ASPs. If done correctly, only the client networking component and the LSM need to know about ASPs.

Volumes

A volume is a complete subtree of a file system. Volumes may contain files and directories. Volumes may not be mounted in other volumes. A volume is a logical grouping of files within the file system and is the unit of replication across servers. An application will reside in a single volume. Two applications will never share a volume.

Volumes are uniquely identified by a 32-bit volume identifier. Each volume additionally has an 8-bit version number. This version number is incremented each time any file within the volume changes. (See supporting upgrades, below). Note that the volume id is globally unique. If two ASPs provide volumes with the same volume number (and version), they have identical contents.

A volume may be replicated on any number of servers. Each SLM server contains a map describing the application servers that currently provide each volume. This global replication of this table is acceptable because volumes are added or moved infrequently.

Identifying Files
Files and directories are uniquely identified by the pair (volume id, file number). This tuple is called a file id. Volume id and file number are each 32-bit signed integers. Negative values for both volume id and file number are reserved for special purposes, leaving us with 2^31 possible volume IDs and 2^31 possible files per volume.

Finding an Application Server for a Volume
The SLM will tell the client which application servers currently provide each volume. It may be necessary for the client to periodically poll the SLM to get up-to-date information about the state of the application servers. The License and Subscription Manager on the client will keep track of the currently subscribed applications and the application servers for each of these applications.

Directories
Directories are specially formatted files that are used in a special way by the file system. They are identified by file ids, just like other files. From a client-server point of view, they are read by the client in the same way as other files. Directories contain arrays of entries with the following format:
(volume number, file number, flags, length, filename)

The volume number and file number are 32-bit signed integers. The flags are 32-bits of flags. The length is 16 bits and is the length of the filename in bytes. The filename is a non-NUL terminated Unicode string. The structure is padded with enough Unicode NUL characters to make the structure a multiple of 32 bits long. The next directory entry begins on the next 32-bit boundary.

The access token is not part of the directory, as a single access token is required to access all files in a particular volume.

The volume number is required so that the the client can construct a local directory for the root of the directory structure in the same format as other directories (see filename parsing below). It also helps to provide a sanity check.

Accessing Files
Assuming that a client has a file-id for a file that it wishes to access, the following client-server actions must be supported:

For stat-like information on the file, we need a GetFileMetadata() interface. The client would provide the file id it is interested in and the proper access token for this file. The server will either return the metadata for the file or an error condition (like access token expired or incorrect access token.) The metadata contains the standard Windows metadata information, including file length and file access times.

On a file open (CreateFile in Windows terminology), we need to verify that we have access to the requested file. This is probably best accomplished by calling GetFileMetadata and verifying that we can get the metadata. This way, we can fail file opens gracefully if we don't have an access token.

On reads (and writes, when we support them), the client will send the file id and the access token to the server along with an offset and a length for the read and write. The server will respond with the data. Note that the same mechanism will be used for reading both files and directories.

Pseudodirectories
For those parts of the eStream file system name space that do not belong to any volume (such as the root of the file system), the client must construct appropriate directories based on the currently installed applications. This is to support filename parsing starting at the root of the directory. For example, if the client has word installed with a root of /Worddir and it is volume number 3 and Photoshop installed with a root of /Photodir and is volume number 4, the client would construct a directory for the root of the entire file system containing
File name, Volume number, file number
"Worddir", 3, 0
"Photodir", 4, 0

(The file numbers are both zero here because 0 is the index of the root directory of each volume, and these are the mount points for each volume.)

When new applications are installed, the root of the file system would have to be updated to reflect the newly installed apps.

Filename Parsing
Filename parsing is handled one element at a time, starting at the root of the file system. Parsing one path name element involves reading the parent directory's contents (from the

cache or the app server), searching it for the file matching the next path element's name, and getting the appropriate file id so it can do further lookup.

Volume Versioning... Without File Versioning

We can provide volume versioning and incremental volume updates without versioning each file in the file system. When a new volume is to be provided, we can append any new or changed files as new files in the volume, with new volume IDs that weren't already present. If a directory's contents have changed, then a new version of this directory will be built, with a new file number. This process will proceed from the leaves all the way to the root of the file system, eventually resulting in a new root. The old versions of things would still be available for old clients to access, but clients wishing to access the new version will simply start at the new root, and would thereby get to a consistent picture of the volume. Any file or directory that has not changed from the old version to the new one need not be replicated, and will be referenced by its old file number. (I.e. newly reconstructed directories will contain the old file number for any files that haven't changed.)

If we reserve the first 256 file ids for the root directory, then the version number can be the same as the file number for the root directory.

Note that if we decide that the complexity of this approach is too high, this does not preclude always creating a new volume from scratch for each update.

Constructing File IDs

It is the job of the builder to produce the volume file to file id mapping and to construct all of the directories. Because directories are files identified by file id, this process must begin at the leaves of the volume and proceed to the root.

Note that constructing a new changed volume will consist of finding the diffs between the two volumes and producing some new directories. Changed or newly added files will get new file numbers, leaving the old ones around. Note that any directory that has had any descendents changed must be reconstructed with the new file numbers, and the new directory will get a new file number. This process will proceed to the root of the volume, which will receive a new file number.

Server Failover

All app servers for a particular volume must share the same mapping of file ids to file, so server failover is trivial. There might be a performance impact if the new app server doesn't have the requrested file in memory.

Writing Files into the Application Install Directories

Two approaches have been discussed for the problem of applications that want to write files to their install directories. First, this can be handled wholly inside of the eStream file system. The cache manager could allow writes to files handled by the eFS, but these writes would not be written back to the server. Instead, they would simply be written to

the eFS cache and marked non-purgeable. This approach's primary advantage is that it does not rely on a file spoofer.

The other approach is to use the file spoofer to spoof some accesses to the z: drive. Any open for read/write access would cause the existing file (if any) to be copied to a location on the c: drive, and the file spoofer would then redirect the open to the newly created file. The file spoofer would have to keep track of any file created via this copy-on-write mechanism and redirect all future accesses to the copy. There are some issues to this approach. For example, it is extremely wasteful when files on the z: drive are opened for read/write access but are never actually written. However, it does help reduce the complexity of the eFS cache, and is trivial to implement if we have to do c: to z: file spoofing anyway.

In either case, to support the creation of new files in an application's install directory, it must be possible to modify the contents of directories in the cache.

If we don't use the file spoofing approach, there is the issue of how we support written files when we move to a newer version of a volume. It would probably be necessary to walk the cache and make sure that each written file gets placed in the appropriate place in the new volume version. This is likely to be non-trivial, because we need to have full information about the location of each modified file in the file system tree, and would need to download enough of the new volume directory structure to place these modified files there.

64-Bit File Access?
One question we should answer is whether we will support file sizes greater than 2 GB on the eStream file system. I'm inclined to say that such support isn't a requirement for the 1.0 product, but I also think that the implementation and verification complexity of 64-bit file access on the file system is low enough that we might want to consider building it in anyway.

Simplifications
We could preclude the possibility of an application consisting of more than one volume.

Future Possibilities
Epicon seems to make a big selling point of their technology involving "self-healing" of damaged application files. Such support could be provided by computing checksums on files in the cache. Whether or not we want to support this is an open question. My feeling is that it's something we should leave out of 1.0.

Outstanding Issues
Cache organization has not been addressed.

Finding and downloading the app install block has not been addressed.

Security in a multiuser system has not been addressed.

eStream File Spoofer Low Level Design

Curt Wohlgemuth
Version 2.0
[redacted]

Functionality

The eStream file spoofer is a kernel-mode driver responsible for redirecting file accesses **from** local file systems **to** the eStream file system driver. It is implemented as a file system filter driver that traps all IRP requests to the file system device handling drives that must be spoofed, and redirects these requests to the EFSD.

Data type definitions

The file spoofer will understand entries in the "file spoof database" as they have been identified by the eStream builder and installed by the app install manager, but these are not defined by this component.

Entries in this spoof database will have the following entries:

- original (fully qualified) path name of file: this resides somewhere on a local disk of the client machine
- new (fully qualified) path name of the file to spoof to: this resides on the eStream file system drive

The spoof database will reside in the registry, so it can be persistent across reboots, and so the file spoofer need not open a file to load it.. As applications are installed on a client machine, the Application Install Manager will load new spoof entries into the registry, and inform the file spoofer that it must reload this database. Similarly, when an app is uninstalled, the AIM will remove spoof entries from the registry, and inform the file spoofer.

This proposes that the each spoof entry is a separate name/value entry under a single key in the registry:

- Name: the original filename
- Value: the new filename

Interface definitions

The eStream file spoofer is called by two components:

1. The eStream client start service, which will start and stop the file spoofer

2. The AIM, which will inform the file spoofer to reload the spoof database from the registry

The interfaces called by both of these user-mode components will be in the form of DeviceIoControl() calls. The following IOCTL codes will be defined for use by callers of the file spoofer:

```
IOCTL_FS_START_SPOOFING
IOCTL_FS_STOP_SPOOFING
IOCTL_FS_RELOAD_SPOOF_DB
```

Starting spoofing

The input buffer for this IOCTL should supply the name of the registry key containing the spoof entries as values. The output buffer for this IOCTL is ignored and should be NULL.

This will return either STATUS_SUCCESS, or an error return status if something goes wrong. It causes the file spoofer to read the spoof registry entries, and load up each entry into memory.

Note that starting spoofing is currently identical to reloading the spoof database.

This is called by the eStream client start service on startup.

Stopping spoofing

The input and output buffers for this IOCTL are ignored and should be NULL. This will return either STATUS_SUCCESS, or an error return status if something goes wrong. It causes the file spoofer to clear its memory image of the spoof database.

This is called by the eStream client start service on shutdown.

Reload spoof database

The input buffer for this IOCTL should supply the name of the registry key containing the spoof entries as values. The output buffer for this IOCTL is ignored and should be NULL.

This will return either STATUS_SUCCESS, or an error return status if something goes wrong. It causes the file spoofer to read the spoof registry entries, and load up each entry into memory.

Note that this is currently identical to starting the spoof database.

This is called by the AIM when a new eStream app is installed.

Component design

The file spoofer will have these major tasks:

- Track the following data:
  - all current valid file spoof entries
  - spoof entries by filtered file system
- Filter native file system drivers for local drives, intercept all IRP_MJ_CREATE and FAST_IO_QUERY_OPEN requests, and for spoofed files, change the file-name of the FileObject associated with these requests.

Data structures

The file spoofer needs to be able to quickly look up a filename in the in-memory spoof database. The current design will use a hash table, whose size and hash function will be tuned as we get experience with real applications.

Adding or deleting entries from the hash table will by synchronized using a global resource.

Algorithms

Here are basic algorithms for these steps.

Load spoof database

This reads all the name/value pairs under the registry key which holds the spoof entries, loads them into a temporary hash table, then points the real hash table to this one.

```
traverse the registry until the input registry key is opened, using ZwOpenKey()
if not found
    return error
if no name/value pairs exist in this key
    return "no data"

for each name/value pair found with ZwEnumerateValueKey()
    build a hash node for this and insert it into temp hash table
    if the drive letter for the old filename entry is one we're not currently filtering
        put this drive letter on new drive list

acquire the hash table resource exclusively
point the global hash table head pointer to the temp hash table
release the resource

for each drive letter on the new drive list
```

```
        look up FS device for this drive
        if we really aren't attached to it
            attach self to this FS device as filter driver

    free the old hash table
    free the drive list
    return success
```

Stop spoofing

```
    acquire hash table resource exclusively
    free the global hash table
    detach self from all filtered FS devices
    release hash table resource
```

Trap Create and QueryOpen requests

```
    acquire the hash table resource shared
    if hash table head pointer is non-NULL
        look up input filename in hash table
    release hash table resource
    if filename not found in hash table
        send I/O request to original target FS driver
    else
        free memory of existing file name in input FileObject
        allocate memory for new, spoofed filename, copy into this memory
        send I/O request to eStream file system driver
```

Testing design

Unit testing plans

The file spoofer will be tested as a standalone component, apart from the rest of the eStream client. A driver test program will be written to test all functionality and corner cases. This includes filtering all FSDs active for a client system, and multiple drives handled by a single FSD.

Stress testing plans

The file spoofer should be able to work, with little or no performance cost to the system as a whole, even when the attached FSDs are under heavy load. Some stress testing will be done in this fashion.

Coverage testing plans

If we come up with a method for measuring coverage for kernel-mode components, we'll do so for the file spoofer as well.

Cross-component testing plans

Not clear if anything need be done here outside of the standard execution of the eStream client.

Upgrading/Supportability/Deployment design

I don't see any upgrade/compatibility issues for the file spoofer. For supportability, there will be a good debugging strategy and sufficient error message return codes for the caller.

Open Issues

This is a list of issues that need to be further investigated or revisited during implementation.

1. We will need to experiment with the hash table to tune it for fast lookup. It's possible that we may need to replace the hash table with a faster lookup algorithm.

SCENARIO: Install a subscribed application

LEGEND: use of specialized fonts in material below:
Bold indicates block or entire scenario
Bold italic indicates argument or return interface between blocks or scenarios
Italic designates meta-info about scenario

BACKGROUND: ways of invoking this scenario

- ⇨ User subscribes to a new application or requests upgrade of existing app.
- ⇨ User starts using a new client.
- ⇨ Client learns that a new application is available.

DISCUSSION: Nature of the **AppInstallManager**
The **AppInstallManager** block may not be a generic module installed as part of the client installation of eStream; instead, it may be a unique executable associated with each particular application. Either way, it is expected that there is certain basic functionality associated with this code; that functionality is described below.

DISCUSSION: Need for checking license at install time
The scenario below describes that the application's license is checked at install time. This may need to be discussed further. Two reasons for license check at install time are:

(1) may be required by the software vendor's licensing model, and
(2) allows eStream client to record which ASP subscription to use when app invoked.

AppInstallManager: invoked w/***DRM server name*** & ***application serial number***

- ⇨ *ACTION*: Establish connection, get license.
    - o Perform **CheckLicense**, which involves asking **ClientNetworkInterface** to send ***check-license*** message including ***application serial number*** to ***DRM server name*** (securely) & getting back ***response***.
        - If response == ***License ok***, then ***App Server name*** & ***session id*** returned, along with ***application nickname*** and ***upgrade flag***. If upgrade flag ***set***, ask **ClientUI** to display ***message*** advising user about upgrade & to solicit ***response*** concerning whether user wants to continue with current version. If response ***affirmative*** or if upgrade flag ***not set***, continue with next *ACTION*. Else, return ***status*** to **AppInstallManager** caller.
        - If response == ***License not ok***, then ***reason not ok*** returned. Some possible reasons license may not be ok include:
            - app license ***expired***. Ask **ClientUI** to display ***message*** advising user about license status & suggesting that user go to the ASP web server & renew license. Return ***status*** to **AppInstallManager** caller.
            - app not accessible (DRM server did not respond, DRM server indicated that application no longer supported, etc). Ask **ClientUI** to display ***message*** advising user that app not

available & suggesting that user go to the ASP web server for more info. Return ***status*** to **AppInstallManager** caller.

- ⇨ *ACTION*: Get application installation information.
  - Perform **GetAppInstallBlock**, which involves asking **ClientNetworkInterface** to send ***get-install-info*** message including ***session id*** to ***App Server name*** (securely) & getting back ***response***.
    - If response == ***success***, then pointer to allocated ***AppInstallBlock*** data is also provided. Control continues with next *ACTION*.
    - If response == ***failure***, then ***status*** returned to **AppInstallManager** caller.
  - Please note that if the **AppInstallManager** is an application-specific program, it may not request the entire ***AppInstallBlock*** contents at once.

- ⇨ *ACTION*: Check if application already installed.
  - Perform **CheckIfAppAlreadyInstalled**, which involves using relevant info supplied as part of ***AppInstallBlock*** to check for the presence of certain registry entries and/or files. This is expected to detect both previous estream & non-estream installations in effect. If app ***not already installed***, continue with next *ACTION*. If app ***already installed***, ask **ClientUI** to display ***message*** advising user about this & to solicit ***response*** indicating whether to continue with current installation. If response ***negative***, then return ***status*** to **AppInstallManager** caller. If response ***affirmative***, continue with next *ACTION*.

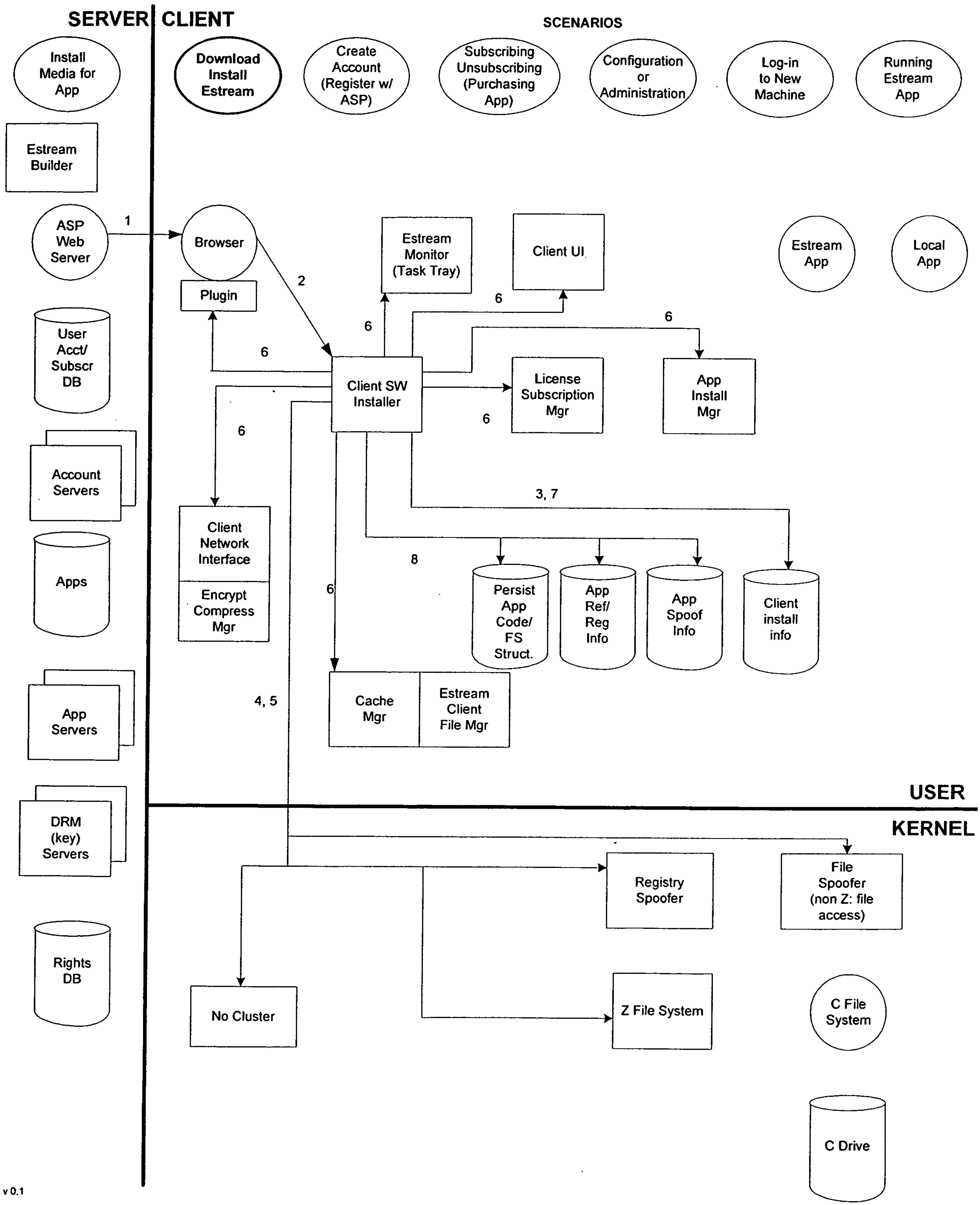
- ⇨ *ACTION*: Parse ***AppInstallBlock*** info, Set up client to make app ready to run
  - *ACTION*: Handle non-Z: file association information in ***AppInstallBlock***.
    - Pass ***ptr*** to non-Z: file association data in ***AppInstallBlock*** to **InitializeFileSpooferData**, which checks client's system against relevant files (creating any non-Z: files that need to exist initially) and which passes that ***ptr*** along with ***filespoofer-data*** designation and ***application nickname*** to **eStreamClientFileMgr** for storage in appropriate area on client.
  - *ACTION*: Handle registry information in ***AppInstallBlock***.
    - Pass ***ptr*** to registry data in ***AppInstallBlock*** to **InitializeRegistrySpooferData**, which modifies client's registry as appropriate and which passes that ***ptr*** along with ***registry-data*** designation and ***application nickname*** to **eStreamClientFileMgr** for storage in appropriate area on client.
  - *ACTION*: Handle application data in ***AppInstallBlock***.
    - Pass ***cache-data*** designation, ***application nickname***, & ***ptr*** to initial application cache data in ***AppInstallBlock*** to **eStreamClientFileMgr** for storage in appropriate area on client.
    - Pass ***profile-data*** designation, ***application nickname***, & ***ptr*** to initial profile data in ***AppInstallBlock*** to **eStreamClientFileMgr** for storage in appropriate area on client.

        - For any application files to be preinstalled, pass ***file-data*** designation, ***application nickname***, ***application filename***, & ***ptr*** to file data in ***AppInstallBlock*** to **eStreamClientFileMgr** for storage in appropriate area on client.
    - *ACTION*: Perform other application-specific activities as desired.
        - If the **AppInstallManager** is generic code, then there is an interface to download an extra executable to do additional activities. If the **AppInstallManager** is code specific to the application, optional extra activities are included in that executable.

⇨ ACTION: Record app installation on client system
- Have **eStreamClientFileMgr** record ***application nickname***, ***DRM server name***, & ***application serial number*** in database of apps installed on client.

⇨ ACTION: Return status to **AppInstallManager** caller.

Scenario 1: Initial install



eStream 1.0 License Subscription Manager (LSM)

Omnishift Technologies, Inc.
Company Confidential

Functionality

This component is a COM Server executable.

The LSM manages the users subscriptions to the different ASP accounts. It is part of the client component downloaded on a client machine. The LSM starts running when the client component starts running and is always active when the client component is running. Users on a given machine establish a connection with the ASP account servers from which they have subscribed applications. Users can add and delete the applications that are subscribed from the ASP accounts. The LSM makes the appropriate calls to the account servers to perform those actions. It gets serial numbers for the applications that are being subscribed and deletes them for the applications being un-subscribed (which are all part of the ASP ID Block). When the users start running any of the subscribed eStream applications, the eStream file system first queries the LSM before servicing any requests. The LSM in turn gets the appropriate access tokens from DRM servers along with the identities of application servers that can be used to run the applications. It uses the client identification (serial number) obtained when the connection to the ASP was made. At the same time, the LSM can decide to cache the access tokens and the identities of the application servers and decide to serve them directly from its cache. The eStream Cache Manager informs the LSM when applications start and end. The LSM keeps track of when access tokens are expiring and can request for additional access tokens when applications are running and the current one is about to expire.

Data type definitions

The global data managed by the LSM includes

1. The ASP ID Blocks which are obtained when the user on the machine establishes a connection with an ASP from which the user has subscribed applications.

Field Name	Type
ASP ID	GUID
ASP NAME	BSTR
ASP URL	BSTR
ASP IP	DWORD

2. The ASP Subscription Blocks are created when the user establishes an account with the ASP service. These blocks enables secure logon to the ASP Server.

Field Name	Type
USER ID	GUID
USER NAME	BSTR
USER HASH PASSWORD	BSTR

3. Application Subscription Blocks are created for every application that is subscribed. These blocks are created when the application subscription is started and are updated when the application is run.

Field Name	Type
APPLICATION ID	GUID
APPLICATION NAME	BSTR
RATE	CURRENCY
PERIOD	INTEGER

4. The access tokens and the identities of the applications servers that are obtained from the DRM servers when the user tries to run the applications.

Field Name	Type
TOKEN ID	GUID
APPLICATION ID	GUID
EXPIRATION	DATE
TOKEN SIZE	DWORD
TOKEN DATA	BYTE *

Interface definitions

Subscription Management

Subscription management is the main interface between the Client UI control panel and the License Subscription manager. Tables containing lists of Application Service Providers and Subscribed Applications are managed using the Subscription manager interface.

ILicenseSubscritpionManager::IDispatch

The LSM exposes the following set of APIs to the client UI.

BOOL SubscribeApp(GUID & ASPId, GUID & AppID, LicenseInfo)

This routine in turn will call the App Install Mgr to install the application on the client machine. This will return a Boolean stating success or failure.

HRESULT UnsubscribeApp(ASPId, AppID)

This routine will NOT implicitly uninstall the application. Applications must be explicitly uninstalled. This will return a Boolean stating success or failure.

HRESULT GetNextAppID(GUID & AppID)

This routine will return a pointer to a list of subscribed applications on the client machine.
The LSM exposes the following set of APIs to the eStream file system.

HRESULT CheckAccess(Path, &Root)

The LSM establishes a co-relation between the Path and the AppID by querying the App Install Mgr. This routine in turn may contact the DRM server for appropriate access tokens. This will return a Boolean stating success or failure. At the same time root will get set to the head of the path that identifies the application so that the file system can use the same access token for everything under "root".

BeginApp(AppID)

To indicate the start of an application.

EndApp(AppID)

To indicate the end of the application.

The LSM makes the following API calls.

1. InstallApp(ASPId, AppID) to the App Install Mgr to install the subscribed applications.
2. GetAppId(Path, &Root) to the App Install Mgr to get the AppId from the Path. "Root" is explained above.

The LSM also sends messages to the account server for subscribing and unsubscribing applications and to the DRM server for getting access tokens. When a user goes to a new machine and installs the eStream client, the LSM obtains the subscription information from the account server when the user first establishes a connection with it.

ISubscriptionManager

ISubscription Methods	Description
SubscribeApp	Subscribes an eStream Application.
UnsubscribeApp	Un-Subscribe an eStream Application
CheckAccess	Check to see if an eStream application is subscribed
BeginApp	Begin an eStream Application
GetNextApp	Get the next application the ASP Supplies
GetNextSubscribedApp	Get the next Subscribed Application
GetLicenseInfo	Get the license info for the application.

HRESULT ISubscriptionManager:: SubscribeApp

```
HRESULT SubscribeApp(
    GUID AppID,
    GUID AspID,
    BSTR *licenseInfo,
   );
```

Parameters

AppID
: [in] Identifier for application to be subscribed.

AspID
: [in] Identifier for the ASP service that the application is going to be subscribed to.

licenseInfo
: [out] License info block for the subscribed application.

Return Values

Returns NOERROR if successful, or an OLE-defined error value otherwise.

Remarks

This function will return an error is a user attempts to subscribe an application that is already subscribed.

HRESULT ISubscriptionManager:: UnSubscribeApp

```
HRESULT SubscribeApp(
    GUID AppID
       );
```

Parameters

AppID
[in] Identifier for application to be un-subscribed.

Return Values

Returns NOERROR if successful, or an OLE-defined error value otherwise.

Remarks

This function will return an error is a user attempts to un-subscribe an application that is not subscribed.

BOOL ISubscriptionManager::CheckAccess

```
HRESULT CheckAccess(
    GUID AppID
       );
```

Parameters

AppID
[in] Identifier for application to be checked for access

Return Values

Returns TRUE if application can be access, FALSE otherwise.

BOOL ISubscriptionManager:: BeginApp

```
HRESULT BeginApp(
    GUID AppID
      );
```

Parameters

AppID

[in] Identifier for application to be started

Return Values

Returns S_OK if application can be started, E_NOACCESS if the application is not subscribed.

BOOL ISubscriptionManager:: GetNextApp

```
HRESULT GetNextApp(
    GUID AspID
    GUID AppID
      );
```

Parameters

AspID [in] ASP Account ID to check for application

AppID

[out] Identifier for application to be queried for ID this will be null when the list of applications runs out.

Return Values

Returns S_OK if application can be started, E_NOACCESS if the application is not subscribed.

Token Management – Overview

Token management is the other major function that the eStream License Manager provides. These tokens are requested from the eStream Server every time an eStream application is started and release when an eStream application is terminated. Access tokens are issued for a finite period and renewed on a periodically depending on their expiration timestamp.

Token Management – Cache Manager Interface

Token management is the service that the License Manger provides to the cache manager. A table will be used store access tokens that the Cache manager uses. The ITokenManger interface provides access to tokens.

ITokenManager::IDispatch

The LSM exposes the following set of APIs to the Cache Manage.

HRESULT ITokenManager::GetToken

```
HRESULT GetToken(
    GUID AppID,
    GUID AspID,
    DWORD expires,
    GUID & TokenID,
    DWORD Tokensize,
    BYTE * Tokendata
   );
```

Parameters

AppID

[in] Identifier for application to be subscribed

AspID

[in] Identifier for the ASP service that the application is going to be subscribed to

Expires

[out] Time interval for which the token is valid

TokenID

[out] Token ID

TokenSize

[out] Size of the token data

Tokendata

[out] token data

Return Values

Returns S_OK if successful, an expired token returns an error.

Remarks

Acquire a token for the License manager.

HRESULT ITokenManager::ReleaseToken

```
HRESULT ReleaseToken(

    GUID TokenID,
   );
```

Parameters

TokenID
: [in] ID for token to be released

Return Values

Returns S_OK if successful, an expired token returns an error.

Remarks

Release a token for the License manager.

Token Management – Client Network Interface

Tokens are provided by the eStream server.

IServerTokenManager::IDispatch

```
HRESULT GetToken(
    GUID AppID,
    GUID AspID,
    GUID ClientID,
    GUID UserID,
    DWORD expires,
    GUID & TokenID,
    DWORD Tokensize,
    BYTE * Tokendata
   );
```

Parameters

AppID
[in] Identifier for application to be subscribed
AspID
[in] Identifier for the ASP service that the application is going to be subscribed to
ClientID
[in] Client Id
UserID
[in] Application ASP account User ID
Expires
[out] Time interval for which the token is valid
TokenID
[out] Token ID
TokenSize
[out] Size of the token data
Tokendata
[out] token data

Return Values

Returns S_OK if successful, an expired token returns an error.

Remarks

Acquire a token for the License manager.

HRESULT ITokenManager::ReleaseToken

```
HRESULT ReleaseToken(

    GUID TokenID,
   );
```

Parameters

TokenID
[in] ID for token to be released

Return Values

Returns S_OK if successful, an expired token returns an error.

Remarks

Release a token for the License manager.

BOOL RenewToken(

```
    GUID TokenID,

);
```

Return Values

Returns S_OK if successful, an expired token returns an error.

Remarks

Renew a token a token for the License manager.

Component design

The License Manger communicates with four other logical units inside the eStream Client.

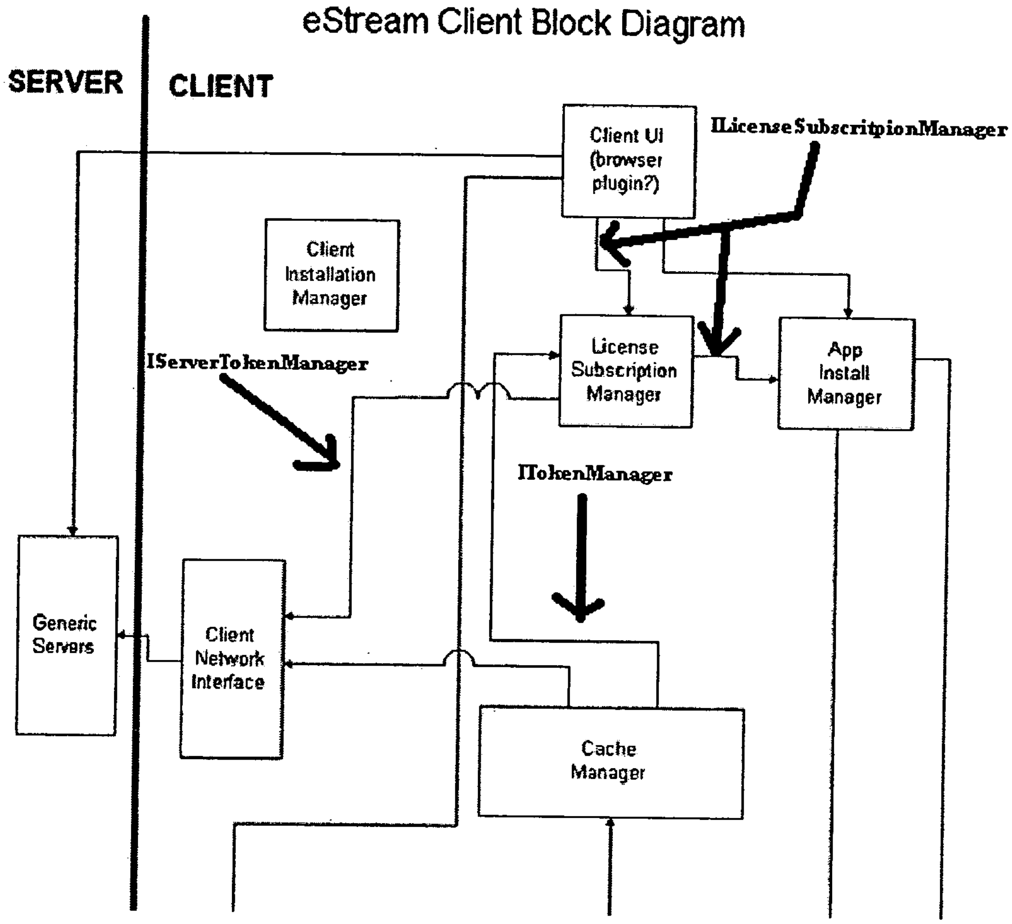


Figure 1 LSM interfaces

This component may be a COM server component. We may decide to implement some of the functions of this unit as an in process DLL that will be access though COM interfaces.

The License Manager communicates with four other logical units in the eStream Client. The interface with the Client UI control panel is through the ILicenseSubscritpionManager. This interface provides a complete list of all ASP accounts, subscribed applications, and accounting information to the Client UI control panel.

The interface with the App Install manager provides lists of application files when a new application is subscribed. These lists are stored in a database table. When an application is started access tokens are requested for the files that are part of the subscribed application.

The interface with the client network provides a connection to the eStream server that will supply the application file binaries to the eStream Client. The function of the LSM is to request lists of access tokens.

Threading Model

In order to service token requests and present application subscription information to the Client UI in a timely manner the License Subscription Manager will need to make use of multi-threading. Currently three threads are planed to fulfill the design requirement of this component. The main thread will satisfy command requests from the Client UI and Cache Manager, and App Install Manager. A separate thread will be spawned when the License Subscription Manager starts to handle Access Token Renewal. A new thread will start for every access token requested or renewed by the Cache Manager.

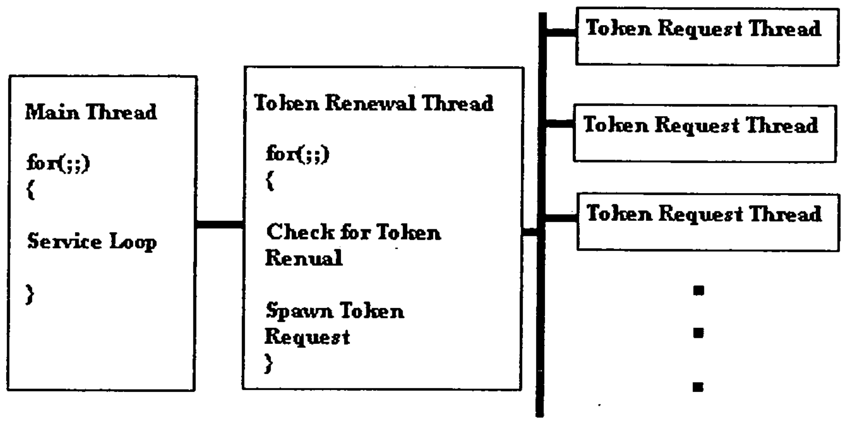


Figure 2 LSM Threading Design

The threads that provide License Subscription services will use Win32 SDK Event semaphores to signal to each other event notifications such a token renewal, network timeouts and token denial.

Main Thread

The main thread provides the interface support for the ILicenseSubscritpionManager and ITokenManager interfaces. When the main thread begins a worker thread is started that clears the token table by releasing and tokens that remain from the last instance of the License Subscription Manager. The token renewal thread sleeps on a timer waiting for an Access token to reach expiration.

```
/*
This is a psuedo code example of how the LSM main Loop will look like.  The
complexity of a COM implementation of this program unit means that the real code will
look very different from this code.
*/

LSMMainThread()
{
        Clear_Token_List();
        Start_Token_Renewal_Thread();
        For(;;)
        {
                Wait_For_Command();
                Switch(CommandType)
                {
                        Case SubscriptionManagerRequest:
                                Service_Request();
                        Break;
                        Case AccessTokenRequest:
                                Get_Access_Token();
                        Break;
                        Case AccessTokenGranted:
                                FireAccessTokenGranted();
                        Break;
                        Case AccessTokenDenied:
                                MessageBox(No Access Token);
                                FireAccessTokenDenied();
                        Break;
                        Case AccessTokenExpired:
                                MessageBox(Expired Access Token);
                                FireAccessTokenExpired();
                        Break;
                        Case Shutdown:
                                Release_Access_Tokens();
                                Terminate_Token_Renwal_Thread();
                                Return;


                }
        }
}
```

Token denial Policy and token expiration policy are two of the most critical issues that the License Subscription manager must handle. The policy for a token denial is to prevent

the user from running the subscribed application. The policy for token expiration is more difficult. Currently the plan is to nag the user into renewing their expired subscription using message boxes. We may move to some other policy as the License Manager develops.

Token Renewal Thread

The token renewal thread is responsible for maintaining the current list of tokens and requesting renewal for each token as it expires. Each time a token expires a new Token Request Thread is started to access the Cline Network Interface for a new Access token from the eStream Server.

```
TokenRequestThread()
{
	InitialIizeTokenTable();

	For(;;)
	{
		WaitForMultipleEvents();
		Switch(Event)
		{
			case TimerPop:
			Break;
			case TokenRequest:
			Break;

			case TokenGranted:
			Break;

			case TokenExpired:
			Break;

			case TokenRefused;
			Break;

			Case Shutdown:
				Kill_Network_Threads();
				Release_Tokens();
				Clean_Up_TokenTable();
				EndThread();
			Break;


		}
	}
}
```

Testing design

Unit testing plans

Stress testing plans

Coverage testing plans

Cross-component testing plans

A Method for Efficiently and Securely Delivering Computer Applications over a Network

Asignee:
Omnishift Technologies, Inc.
2480 North First Street, Suite 150
San Jose, CA 95131
(408) 321-6900

Manuel E. Benitez
10245 Parkwood Drive # 6
Cupertino, CA 95014
(408) 255-8731

Managing a networked computing environment is a daunting task. The laborious process of ensuring that each computer contains a current version of each application is very time consuming. Several solutions exist to help *Information Technology* (IT) departments reduce application management costs and improve the likelihood that each computer has the appropriate version of each application. These solutions fall into three categories: server-based computing, automatic distribution and application servers.

Server-based computing solutions simplify application manageability by running applications on a *farm* of servers along with mechanisms that deliver the output of the application to a *client* machine and send the keyboard and mouse input back to the server. In this manner, server-based solutions give the appearance of providing the appropriate version of each application on every machine. Using a server-based solution, IT departments reduce application management efforts to managing a server farm. Additionally, applications with modest graphical requirements can be delivered across limited bandwidth network connections and be available during business travel or for telecommuting outside the corporate network. The drawback of server-based solutions is the server farm must provide sufficient computational resources to run all the applications requested simultaneously. Doing so, especially during peak demand periods, requires very substantial investments in servers. Providers of server-based solutions claim that server costs are offset by reductions in the computational requirements placed on client machines. In practice, server-based solutions rarely result in the purchase of cheaper, less-powerful clients because users prefer to retain the freedom to use applications not available through the server-based infrastructure.

Automatic distribution solutions address application availability and versioning issues by providing a mechanism whereby client machines can automatically download new and updated applications from a central server. Automatic distribution solutions consist of a mechanism that takes an inventory of the applications on a client machine and compares it against the current application list. When an update is required, automatic distribution solutions leverage the standard application installation and upgrade processes.

Unfortunately, this requires transferring the entire application to the client, a process that can take minutes across a fast network and hours for business travelers or telecommuters. On the positive side, automatic distribution solutions scale more easily than server-based solutions, as a single server can handle many times as many users and applications.

Application server solutions address application availability and versioning issues by placing all applications on a central storage location. Client computers access these applications through a *network file system* that acts like a standard local file system on a client machine, but in fact provides files stored elsewhere on the network. Many clients can access the same copy of the application stored on a network file system and the IT department can easily upgrade or install applications that can then be used on all client machines. The network file solution works well in cases where network bandwidth is high (10Mb/sec or greater) and latency is low. Current solutions, such as NFS developer by Sun Microsystems, also lack robust security features and are intended for use inside secure corporate LANs.

Background

Figure 1 illustrates a basic computer system. The *central processing unit* executes application instructions that request it to perform operations such as addition, subtraction, multiplication, division and moving data between the various system components. The

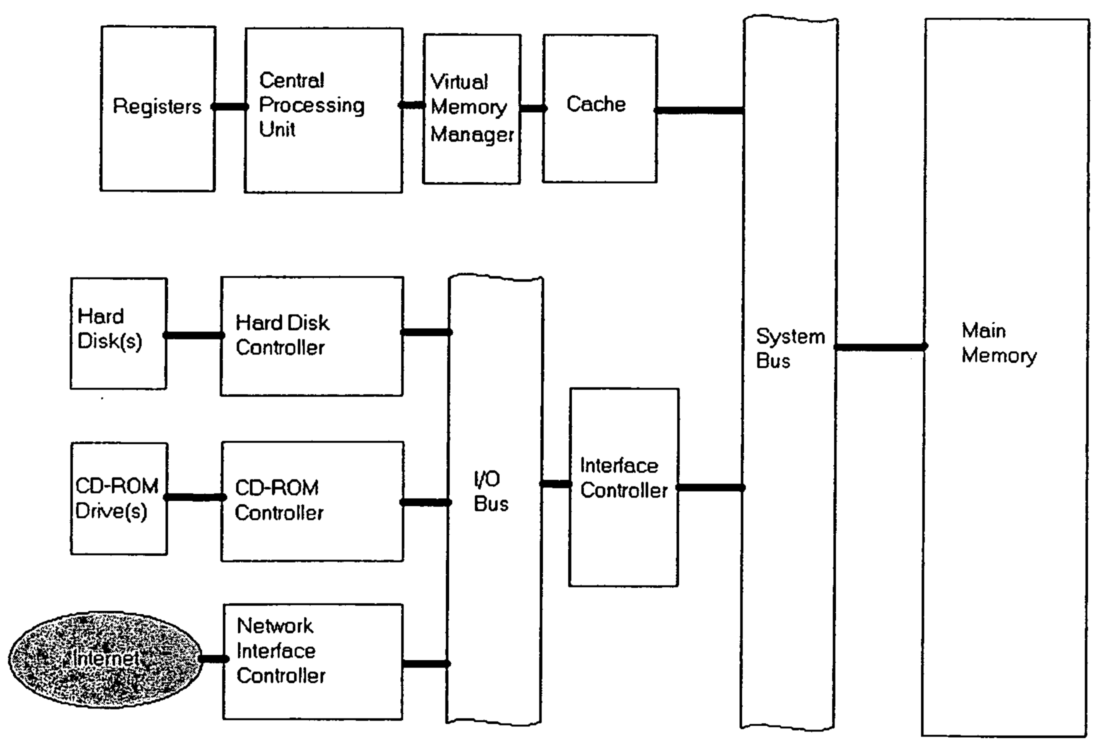


Figure 1: Computer System

central processing unit has two main areas in which it manipulates data: *registers* and *main memory*. Registers are fast but few in numbers. Accessing the main memory takes much longer, but there is much more space to hold data in main memory than there is in the registers. While the central processing unit communicates with the registers directly, its link to main memory and the rest of the is through a communication pipe called the system bus. The system bus coordinates data transfers between system components and operates more slowly than the central processing unit does. Between the central processing unit and the system bus are two special components known as the *virtual memory manager* (whose purpose will be explained presently) and the *cache*. The purpose of the cache is to keep a copy of the most recently referenced data stored in main memory. This is done so that the system bus does not need to be used if these data are referenced again. The cache improves the performance of the system because of a phenomenon called *locality*. Locality dictates that the most recently referenced data are the most likely to be referenced again.

One of the components directly connected to the system bus is the *interface controller*. The interface controller acts as a buffer between the system bus and the slower and more complicated *Input/Output (I/O) bus*. The job of the interface controller is to convey I/O requests from the central processing unit to the *I/O device controllers* and transfer data between I/O device controllers and main memory. Applications running on the computer system are not permitted to communicate with either the interface controller or the I/O device controllers directly. This is because controllers are very sensitive to the timing of events and can easily be put into states where they stop operating properly or start misbehaving so that other components can no longer perform their tasks. The computer system is managed by a special application known as the *operating system*. The operating system is made up of many components. Among these components are a group known as the *device drivers*. The main purpose of a device driver to hide the intricacies of dealing with the I/O interfaces from the rest of the operating system.

I/O device controllers perform the task of controlling the physical or electronic components that make up a device. For example, the *hard disk controller* converts a command to read a particular block of data (called a *sector*) from the hard disk into appropriate levels of electrical current to move the disk's read/write heads to the precise area of the disk in which the sector is located. It also converts the electrical impulses returned by the head's amplifiers into streams of one's and zero's and scans them to determine when the appropriate sector is being read by the head. Once the interface has read the sector and verified through the use of *error detection codes* that it was read correctly, it communicates over the I/O bus to the interface controller and informs it that the sector is ready to be transferred so that it can begin its journey to main memory and, ultimately, the central processing unit.

The *network interface controller* is another component that is commonly found on the I/O bus. Like other device controllers, the network interface controller performs the task of converting commands to send or receive information across an external network connection into the appropriate electrical currents and voltages required to exchange data across the particular type of network that the interface is connected to. The network interface controller also works in conjunction with an appropriate device driver through which applications send and receive data across the network. Because a large part of networking is based on following complex protocols, naming schemes and software-controlled virtual connections, more sections of the operating system are usually located between the network device driver and applications which handle all of the intricacies of splitting long streams of data into shorter ones (called *packets*) which are acceptable to the network. The collection of physical and software components connected together to support network communications is known as the network stack and an instance of one is shown in figure 2. In this illustration, a network interface controller capable of communicating on an Ethernet network is physically connected to an I/O bus using the Peripheral Component Interconnect (PCI) standard. These two components make up the physical or *Hardware* portion of the network stack. The operating system provides a device driver for each of these physical components that are shown at the bottom of the *Software* portion of the network stack. The operating system also provides a component that communicates via the device drivers to implement the Internet Protocol (IP) while another component will use the IP component to implement the Transmission Control Protocol (TCP) and an application might use the TCP component to implement the HyperText Transfer Protocol (HTTP) to implement a web browser.

Having explained the basics of computer systems and network stacks, we turn our

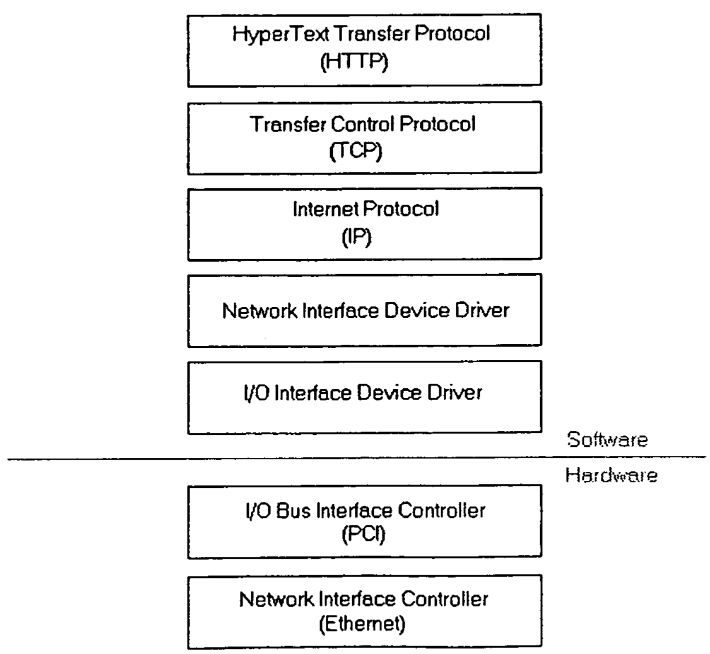


Figure 2: The Network Stack

attention to the virtual memory manager. The central processing unit can reference more main memory than is physically available. The virtual memory manager's task is to mitigate this problem by using a storage device such as a hard disk as an overflow area for data that are not frequently used. This is analogous to keeping documents that are infrequently needed in boxes in the garage. When such a document is needed, its box is brought in from the garage and another box inside the house is selected and taken out to the garage to keep the house tidy. Likewise, the virtual memory system keeps boxes of data called *pages* along with a data structure called a *page table* that keeps track of were each page is currently stored. When the central processing unit needs a piece of data, it sends the *address* of the data to the virtual memory system which quickly determines which page that address is in and consults the page table to determine where that page is located. If the page is in main memory, then the address is modified to location in main memory where the page data actually resides. If the page is not in memory, the virtual memory manager *interrupts* the central processing unit to run another component of the operating system known as the *page fault handler* to obtain the page from the hard disk and copy it to main memory where it can be accessed by the central processing unit. The page fault handler calls on the appropriate device drivers to read the page from the hard disk and copy it to main memory. Before doing this, the page fault handler will usually need to make room for the desired page by selecting another page currently in main memory and writing it to the hard disk so that it can be read back if it is ever needed again.

One of the most fundamental concepts underlying the computer system described here is that *applications* are *data*. They can be wholly or partly stored in main memory or on a hard disk or a CD-ROM disc or, for that matter, on another computer system accessible through the network. Because the central processing unit accesses and manages data in main memory, any part of an application that the central processing unit wants to execute must be brought into main memory. Since there is no difference between applications and data, the virtual memory manager and the cache will handle application components as if they were any other pages of data. One advantage of this is that it allows a computer system to execute an application without having it completely in main memory. The operating system can *map* the application to a set of addresses through the expedient of changing the page table to indicate that the pages comprising the application currently reside on the sectors of the hard disk where the application is stored. When the central processing unit attempts to execute some portion of the application for the first time, the virtual memory manager will interrupt the central processing unit and the page fault handler will copy a page's worth of the application to main memory. The central processing unit can then execute the application instructions in that page without being interrupted until the application strays into a page that has not yet been brought to main memory.

There are numerous advantages to mapping applications via the page table (aka *memory mapping*). The first is that large portions of most applications are never needed except in rare circumstances. For example, very few *spreadsheet* application users make use of *pivot tables*, although a large number of instructions exist solely for the purpose of implementing the pivot table functionality of the spreadsheet application. Bringing all

these instructions into main memory before they are needed would almost always be wasted time and effort. In fact, only a small portion (about 10-20%) of most applications is referenced most (80-90%) of the time. This phenomenon was introduced earlier as *locality*. Memory mapping thus improves performance by reducing the amount of application code that need to be transferred from the hard disk to main memory. Also, since only 10-20% of an application is needed at any given time, memory mapping allows an operating system to simultaneously run several applications while using only the amount of main memory that would be needed to hold one entire application.

Technical Description of the Problem

Locality and memory mapping would then seem like the perfect solution to executing applications across a network. Rather than having the application installed and stored on a computer system's hard disk, it can be kept somewhere on the network and paged in as needed. There are several problems with this approach, however, namely: bandwidth, latency and security.

To handle a request for a page from a hard disk, the page fault handler would determine from the page table which sector on which disk the page was located in. The following steps need to then take place:

1. The read sector command is sent to the disk device driver which will place it on a queue where it will wait until all previous commands for that disk have been sent and the hard disk controller indicates that it is ready to receive its next command
2. The read sector command and the address of the appropriate hard disk controller are passed to the interface controller device driver, which places them on a queue where it will wait until all previous commands for the interface controller have been sent and the interface controller indicates that it is ready to receive its next command
3. The interface controller waits for the I/O bus to become available and transmits the read sector command to the hard disk controller
4. The hard disk controller determines where it needs to position the read/write head and sends appropriate levels of current to the head controller to move the head
5. The head controller and head are physical devices which obey the laws of physics and must accelerate, cross the distance in space towards where the sector is located and then decelerate to stop the head at the appropriate location
6. The disk platter, which is also a physical device, is spinning at a constant number of revolutions per second and the disk controller must wait until the desired sector begins to travel under the head so that the sector can be read
7. The hard disk controller reads each bit of each byte that makes up the sector and its error detection codes and stores them in a small memory buffer, the rate of speed at which this happens is determined by how long it takes the spinning platter to rotate across the length of the sector
8. The sector is then verified by the hard disk controller by examining the sector data and the error detection codes

9. The hard disk controller waits for the I/O bus to become available and transmits a message to the interface controller informing it that it has the requested sector
10. The interface controller waits for the I/O bus to become available and transmits a message to the hard disk controller requesting that it send the sector data over the I/O bus
11. The hard disk controller waits for the I/O bus to become available and transmits the sector data to the interface controller
12. The interface controller places the sector data in a memory buffer
13. The interface controller waits until the system bus is available and then transfers the sector data to main memory
14. The interface controller sends a request on the system bus to the central processing unit indicating that it would like to communicate with its device driver
15. The central processing unit places the request on a queue and, when it is ready, begins to execute the device driver
16. The device driver determines which command has successfully completed, removes it from its queue and informs the operating system that its should execute the disk drive device driver because something that interests it has happened
17. The hard disk device driver determines which command has successfully completed, removes it from its queue and informs the page fault handler that its page is now located in main memory
18. The page fault handler updates the page table to reflect the new location of the page and informs the central processing unit that it can resume executing the application that caused the page fault

This represents a substantial amount of work. Fortunately, most of these operations are completed very quickly given the tremendous computational capacity of a computer. The most time-consuming items are the ones that transpire in the physical domain. Moving the disk head takes about 8-12 msec and waiting for the platter to rotate another 0.2-0.5 msec. Another significant factor is the time spent transferring the page over the I/O bus whose bandwidth is in the 20-80 Mb/sec range. For standard 4 KB pages, this consumes between 0.4 and 1.6 msec of time. Translating this into real time as might be perceived by a human user, if a large application incurs 1000 pages faults (4 MB, average for a 20-40MB application) the system would spend about 12 seconds handling page faults. This would be spread out across the execution of the application with roughly one-third of it happening when the application is initially started. Since large applications usually execute for many minutes, the overall time spent handling page faults tends to be unnoticeable to a user except at the very start of the application or when the system is pushed beyond the point at which the physical memory available can hold the portions of the applications that it is executing. The latter situation is known as *thrashing*, which is characterized by constant disk activity and very little useful progress.

Suppose that the application were to reside on another computer system and the virtual memory manager could access this via the network interface controller. The previous page fault scenario would now be handled as such:

1. The read page command is sent to the appropriate layer of the network stack (most likely the HTTP layer) and would work its way to the network interface device driver which will place it on a queue where it will wait until all previous commands for that network interface have been sent and the network interface controller indicates that it is ready to receive its next command
2. The read page command and the address of the appropriate network interface controller are passed to the interface controller device driver, which places them on a queue where it will wait until all previous commands for the interface controller have been sent and the interface controller indicates that it is ready to receive its next command
3. The interface controller waits for the I/O bus to become available and transmits the read page command to the network interface controller
4. The network interface controller waits for the network to become available and sends appropriate levels of current across the network to send the request for the page
5. The message is received by the computer system that contains the page
6. The page is placed on the network
7. The network interface controller detects the reply and reads each bit of each byte that makes up the page and its error detection codes and stores them in a small memory buffer, the rate of speed at which this happens is determined by the bandwidth of the network
8. The page may have been broken up into a number of smaller packets, in which case the network interface controller waits for each packet to arrive and reconstructs the original page in a memory buffer
9. The page is then verified by the network interface controller by examining the page data and the error detection codes
10. The network interface controller waits for the I/O bus to become available and transmits a message to the interface controller informing it that it has the requested sector
11. The interface controller waits for the I/O bus to become available and transmits a message to the network interface controller requesting that it send the page data over the I/O bus
12. The network interface controller waits for the I/O bus to become available and transmits the page data to the interface controller
13. The interface controller places the page data in a memory buffer
14. The interface controller waits until the system bus is available and then transfers the sector data to main memory
15. The interface controller sends a request on the system bus to the central processing unit indicating that it would like to communicate with its device driver
16. The central processing unit places the request on a queue and, when it is ready, begins to execute the device driver
17. The device driver determines which command has successfully completed, removes it from its queue and informs the operating system that its should execute the network interface device driver because something that interests it has happened

18. The network interface device driver determines which command has successfully completed, removes it from its queue and informs the rest of the network stack that the page arrived until finally the page fault handler is informed that its page is now located in main memory
19. The page fault handler updates the page table to reflect the new location of the page and informs the central processing unit that it can resume executing the application that caused the page fault

This sequence does not appear to be any more complicated than the previous one. Appearances are deceptive because steps 4 through 8 have been understated. Exposing these issues requires an understanding of networks.

A network is a collection of computers joined by a communication link and a common protocol that allows them to successfully transmit messages from one to another. Because the communication link is shared, a computer that wishes to send a message to another must usually wait it turn or ask to be given permission to speak. If this were not so, then most messages would collide with other messages traveling along the network and very little useful communication would take place. As more computers are added to a network, the amount of time that they must wait to send a message relative to the amount of time needed to transmit the message quickly increases. Some relief can be obtained by limiting the length the message each computer can send before it must yield the network to another machine by splitting long messages into a sequence of smaller *packets*. Even then, if too many computers are connected to the network the wait becomes intolerable.

To allow large (*wide-area*) networks to work, the network is divided into many smaller networks called *subnets*. Each subnet is limited to a handful of computers. This makes it easy for one computer to communicate with another computer on the same subnet without having to wait very long. Within each subnet there is a special computer known as a *gateway*. The gateway is special in that it is able to communicate with the world beyond the subnet. When a computer needs to send a message to a computer on another subnet, it sends it to the gateway. The gateway receives the message and decodes it enough to determine who the intended receiver is. The gateway consults a data structure called a *routing table* to determine which of the computers that it can communicate with can forward the message to its intended receiver. The process of receiving a message, consulting the routing table and forwarding the message is called a *hop*. Sometimes, the gateway will receive a message from beyond its subnet addressed to one of the subnet computers. The gateway will realize that it can communicate with that computer directly and forwards the message straight to the destination computer. All network interfaces on the subnet will "see" the message, but only the network interface on destination computer informs its network stack that a message has arrived.

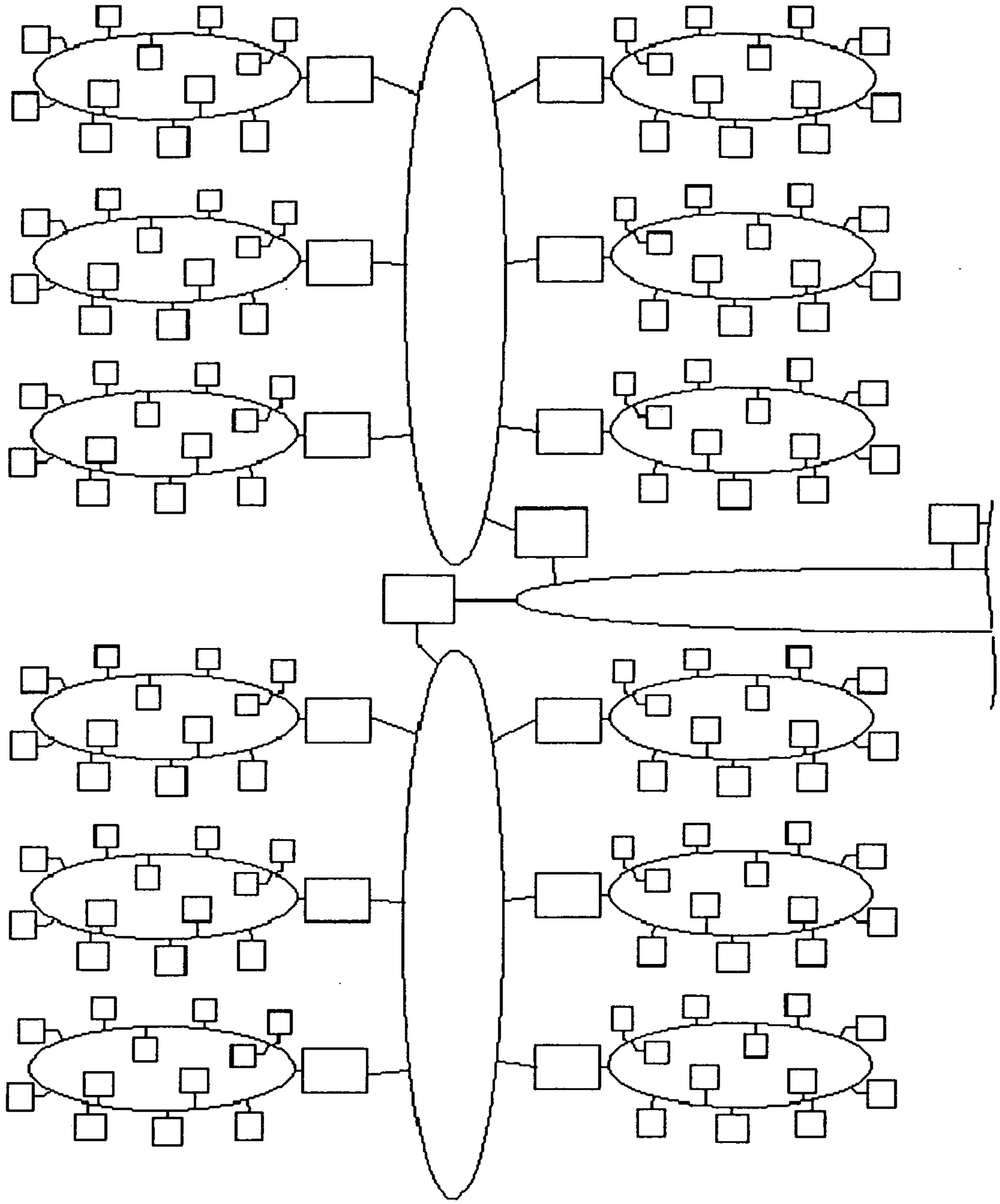


Figure 3: Portion of a Wide-Area Network

Figure 3 illustrates how a large (*wide-area*) network such as the internet can be constructed by subdividing the network into many subnets and linking them together using gateways and other special computers, called *routers*, that do nothing but exchange messages. Using this scheme, a lot of communication transpires in parallel on different subnets so that each individual machine does not have to wait long to send a message on its subnet. The downside of this scheme is that messages might need to cross many subnets to reach their destination. At each crossing, a hop takes place requiring the address of the message to be decoded, a decision to be made about which computer to forward the message along to, and a forwarding of the message on a different subnet. Along the way, different subnets might impose different limits regarding how long each

message can be which may force a gateway or router to split the message up into two or more packets and forward each of them separately.

With this understanding of networks, we return to steps 4 and 5 of the network paging process. If the computer containing the page is on the same subnet as the computer running the application, these steps take little time to complete. Depending on the type of network, its bandwidth and assuming that the distance between the computers is no more than a few hundred meters, the latency of this event is in the 0.1 to 1 msec range. These assumptions hold for small, carefully crafted commercial environments. Step 6 requires the receiving computer to process the request for the page through its network stack, locate the requested page and invoke its network stack to send the page data. The time required to locate the page will likely be similar to the time it takes to obtain a page from a local hard disk. Sending the reply involves another short delay to obtain clearance to use the network. If the network has been crafted properly, as would be the case in a commercial subnet, then the reply will not have to be split up into smaller packets. Step 7 depends on the bandwidth of the network. Assuming commercial, 100 Mb/sec bandwidth, this will take 0.3 msec.

Given a carefully chosen and configured subnet as one might expect to be able to craft in a commercial environment, each page reference over the network would be serviced in no more than 15 msec. The large application that incurs 1000 page faults will spend 15 seconds waiting for pages. This is nearly indistinguishable to a human from the local hard disk and quite acceptable. A commercial subnet environment is also easy to isolate and protect from potential security hazards with *firewall* and *proxy* gateway computers that allow only trusted messages to enter and leave the subnet. This level of security ensures that the application cannot be obtained without permission and that computers on the subnet cannot be improperly controlled by replacing real page reply messages with pages containing a dangerous *virus* or *Trojan horse.*

Consider the case where the computer executing the application and the computer serving the pages are several network hops from each other and connected across subnets whose bandwidth is less than 100 Mb/sec. Each hop incurs a delay of 1 to 10 msec while the message address is decoded, a routing table consulted and the message buffered and re-packaged to send on to the next hop. Some subnets, particularly the ones that reach a residence are physically large (many miles between the gateway and the other computers) and have bandwidths of between 0.5 Mb/sec to 2 Mb/sec. Under these conditions a page request could be expected to take anywhere from 60 msec to upwards of 600 msec to be serviced, or 5 to 50 times the local hard disk page service interval. In this scenario, a large application making 1000 page requests would spend 60 to 600 seconds waiting for pages. This is a very noticeable and unacceptable amount of time. Yet, it is this scenario much more than the previous one that is reflects the environment available to wide-area commercial network and residential users.

Technical Description of the Invention

The invention consists of the following components:

1. an *execution controller:* Run when an application resident on a remote system (called the *server*) is to be launched on a local system (called the *client*). Establishes association between application to be run on the client and its associated files on the server. Handles client side of initial security protocol between client and server.
2. an *application remote file interface:* Handles client interface to accessing files associated with an application that is resident on a remote server.
3. an *application cache manager:* On the client, locally stores previously requested portions of files and file system information associated with an application resident on a remote server. Requests referenced application code and data not currently in the client's cache. Employs existing profiling information to prefetch portions of the application from the server. Collects new profiling information about application to improve client prefetching in the future; this profiling information may also be uploaded to the server for use in improving prefetching performance and to assist in better pre-compression of file data.
4. an *application file server:* Responds to requests by client application cache manager for portions of application's files and directory structure on the server. Transmits compressed information (which may be pre-compressed with nearby data) for better bandwidth utilization. May send extra file data beyond that requested, if that data is expected to be referenced soon.
5. a *file system reference profile processor:* Processes the uploaded sequence of application file references and frequency information. This information, which was collected by the application cache manager, is used in its processed form by the "application stream set builder" in generating pre-compressed file datasets.
6. an *application stream set builder*, used to construct the *application stream sets* that the application file server consults to reply to application file requests.

The execution controller is a small piece of code that resides on the client. The execution controller is given an *argument* indicating which application is to be executed. From the point of view of the client and its operating system, the application is resident locally on the client; the execution controller negotiates with an appropriate server to allow the client to obtain (as needed) segments of the associated application files located on the server.

The execution controller handles the client side of the security protocol between the client and a server; one approach to implementing this security protocol is as follows. The execution controller contains a *security certificate* which uniquely identifies/distinguishes it from every other instance of an execution controller. This certificate has a *private key* that can be used to encrypt any message so that it can be decrypted only with the corresponding *public key* known to the server. Additionally, the execution controller knows the public key of the server but only the server has the private key which can decrypt messages encrypted with it. When starting, the execution controller forms a message indicating which application it is instructed to execute and

attaches to this message a randomly generated key which will be used to encode all subsequent messages between the client and server. The client encrypts this message with its private key and then encrypts the encrypted message with the server's public key. This doubly encrypted message is sent across the network to the server. The server uses its private key to decrypt the message. This has the effect of giving the client a high degree of confidence that only the true server intended to receive the encrypted message views its actual contents. The server then decrypts the message again using the client's public key. This has the effect of giving the server a high degree of confidence that the message was generated by the client. As a result, the server knows which application the client wants to execute and which random key to use in subsequent exchanges with the client.

Upon receiving notice from a client that it wants to execute an application, a server (or set of servers) performs the following actions:

- consults a database which indicates which applications the client is allowed to execute and, if the client is not allowed to execute the requested application, informs the client that it will not be served,
- determines if the application has been updated and, if it has, indicates this to the client, along with information concerning accessing the updated version,
- determines appropriate server location(s) of the desired application,
- checks load on these candidate servers,
- refers the client to the candidate server with the most appropriate load,
- informs the client that it is ready to serve the application.

Upon receiving a reply, the client will either continue with the application execution process, notify the user that it cannot proceed, or interact with the user to determine what action to pursue next, depending on the nature of the reply returned by the server.

If a server accepts the task of serving the application to the client, the execution controller passes the application access request on to the application remote file interface code. This code allows the client to reference file and directory information associated with the remote application as if it resided on a local physical disk device. It uses the network stack to request portions of the application's files and directory structure from the server and borrows storage space from a physical hard disk device on the client to archive this data for future use. The archive storage on the client is managed by the application cache manager, which is another small piece of code running on the client. The invention requires the execution controller, the application remote file interface, and the application cache manager to have been previously installed on the client via traditional software delivery methods.

The client's operating system begins executing the requested application located remotely on a server. The operating system memory-maps the application and begins executing it, with the application remote file interface code obtaining control whenever the client system's page fault handler determines that the application's page is located on the remote disk drive. The page fault handler asks the application remote file interface code to place the appropriate page data in main memory. The application remote file interface code sends a request to the cache manager for the desired data. If the application cache

manager has the data, it is placed in main memory and the application remote file interface code returns control to the page fault handler. If the application cache manager does not have the requested page data, it formulates a message to the server indicating which portions of the remote disk it needs, encrypts this message with the random key that the execution controller produced, and invokes the network stack to send the read message to the server. The requested portion of file data is identified by file name (or some numeric ID) and the pages of the file desired.

The application file server, upon receiving the message, decrypts it with the random key. This gives both client and server confidence that the request was sent by the real client and received by the real server. The server uses the file name and portion of the requested application to lookup or create a reply message. The simplicity of this action is critical to the invention because it is essential that the server respond quickly to any page request. The server makes every effort to index and pre-compute reply messages and to keep them in main memory where they can be rapidly accessed by the server's central processing unit. The response message may contain not only the requested page data, but also several other pages that will very likely be needed by the client in the near future. Alternatively, the client itself may request pages in advance of the application demanding them, by use of its local profile data. The stored response message is also pre-compressed to reduce its length; it is expected to be approximately halved. The response message is encrypted with the random key (this step is not done in advance, since each client sends a different random key, which is used instead of private and public key pairs because they require less time-intensive algorithms) and sent back to the client.

Upon receiving the reply, the client decrypts the message using the random key. This gives the client a high degree of confidence that it is receiving the reply sent by the real server. The client un-compresses the response and parses out the pages returned in the reply. Each page is returned to the application cache manager for future reference. The requested page is placed in main memory and the application remote file interface code returns control to the page fault handler, which allows the client's central processing unit to proceed executing the application.

When the next page fault occurs, there is a high probability that the application cache manager already holds the requested page. The page was either sent by the server on a previous run of the application or was packaged in a previous response to a page request during the current run. This likelihood is because applications have a significant amount of predictability in the order to which they reference sectors on a disk. These patterns can be determined over time by keeping a trace of page requests. In the invention, the application cache manager performs this task. As requests for pages are sent to the cache manager, it notes which pages were previously referenced in a table indexed by the page number. For example, when a request for page 513 of some file is followed by a request for page 1023 of some file, the cache manager records this information in a *page trace table*. The information in this table may be compiled into a message and sent to the server periodically and upon exit of the application. This process, known as sampling, places very little computational demand on the client and the server. The server stores these

tables and uses them after some time to build a new application stream set for the application.

Aggregation and analysis of the uploaded profile data is done by the file system reference profile processor, which is an application executed on a computer system that may be different than the client or server. The following process may be employed by this code to produce trace data used to build application stream sets:

- initialize a two-dimensional table, *a*, from $a[0, 0]$ to $a[s_{max}, s_{max}]$, where s_{max} is the largest page number, to zeroes,
- for every element $t[s]$ in a page trace table, *t*, where $t[s]$ is a valid page add one to $a[t[s], s]$ and
- for every column *c* in *a*, calculate *d* to be the sum $a[c, 0] + a[c, 1] + ... + a[c, s_{max}]$ and, if *d* is not zero, divide every element of the column by *d*.

The effect of this process is to generate a table *a*, where $a[s, f]$ indicates the probability with which page *f* has been measured to follow page *s*. This will be 0 if *f* was never found to follow *s* and 1.0 if *f* was always found to follow *s*. Probability theory dictates that, given a sufficiently large set of page trace tables, the probabilities in *a* will be very close approximations of the actual probabilities.

The process of building a new set of request replies for an application is called building an application stream set. This process takes place on a computer that may be different than the client or the server and takes place at least once before the application is executed using the invention. An application stream set contains:

- a unique name of the application for reference purposes,
- an index table used to quickly determine which reply to return for a given request,
- the set of all possible request replies, each one being a catenation of the actual page requested followed by zero or more additional pages that are deemed by the application stream set building algorithm to be highly likely to be referenced immediately following the first reference, the collection of which is then compressed using a suitable compression algorithm.

The application stream set is built in the following manner:

- instantiate a virtual hard disk drive large enough to contain all of the application's executable and non-executable data and all of the indices (often known as *directories* and *files*) needed by the operating system to properly identify and reference the data,
- install the application on the virtual hard disk using any one of the traditional application delivery models,
- initialize the application stream set to be empty,
- add the unique name of the application to the stream set,
- add an index table to the application stream set containing an entry for each sector in the virtual hard disk drive,
- for each page, *s*, in the virtual hard disk drive:
  - initialize a buffer to be empty,
  - place the page data of *s* in the buffer,

- if aggregated page trace data, a, is not available, skip the following step and go to the compression step,
- perform the following sub-steps:
- initialize set m to contain the pair $\{s, 1.0\}$,
- for some pair $\{s_0, p0\}$ in m, if $p_0 \times a[s_0, s_1]$ is greater or equal to threshold t, and s_1 is not already in m, add $\{s_1, p_0 \times a[s_0, s_1]\}$ to m,
- add a fixed-sized marker indicating the number of s_1 to the buffer,
- add the page data of s_1 to the buffer,
- repeat the previous three sub-steps until no more items can be added to m,
- compress the data in the buffer and add it to the application stream set and
- update the index table entry corresponding to the page to reference the starting location of the just-added compressed data buffer.

The process of building an application stream set must be started without any aggregated trace data since the trace data cannot be collected until there is an application stream set with which to execute the application. The process that is followed is to build an application stream set using the process given without aggregated trace data. This will result in an application stream set that contains only one page per page reply. This application stream set is then used in a controlled commercial subnet environment so that the application will execute with reasonable performance. This environment is used to execute the application under normal conditions for several hours. This will yield enough trace tables to produce the first cut of aggregated trace data that will yield an application stream set that allows the application to execute across a less controlled network environment. This new application stream set can then be used for enough time to collect a much greater set of trace tables which in turn will allow an even better application stream set to be built. This process can be iterated several times.

The most appropriate value of threshold t varies for each different application. Too high a threshold value (near 1.0) will result in responses that contain few pages in each response message and will not improve the performance of the invention over a simple service method. Too low a threshold value (near 0.0) will result in replies that contain too many pages and will require too long to be sent. Such replies will cause the paging response performance of the application to be erratic and noticeable to the client's user. The ideal reply size for the network connection under consideration can be determined via analysis or experimentation. Then the application stream set builder can automatically determine the most appropriate threshold value t using a simple binary search technique. The builder starts with a threshold of 0.5 and builds an application stream set. If the average number of sectors in each reply is greater than that desired, then it subtracts 0.25 from the threshold value and iterates through the build process. If the average number of sectors in each reply is lower than desired, then it adds 0.25 to the threshold and iterates through the build process. The iterative process continues with the amount added or subtracted reduced in half on each iteration. The process ends when either the desired reply size is reached or when a large number, say 100, build iterations have transpired.

Benefits of the Invention

The first benefit of the invention is a dramatic reduction in the perceived paging delay when operating across a network. By choosing appropriately sized request replies that have a high probability of proximate reference, each response returns several useful pages for the latency of one. Compressing the replies to reduce their average length in approximately half effectively doubles the bandwidth of the network. Together, these strategies yield a substantial reduction in the perceived latency. Thus, an unacceptable delay of (say) 60 seconds becomes an acceptable delay of (say) 12 seconds. Additionally, by automatically caching returned data, the invention nearly eliminates the need for network requests on all but the first execution of the application. After an initial, slightly slower than average execution, the application will generally execute with the same paging behavior as if it were traditionally delivered on the client.

Performance measurements collected using an implementation of the invention strongly demonstrate its value. The wall-clock time required to execute the Microsoft Office Word application (bring it up and shut it down) across a 1 Mbps link with the naïve network-unaware approach of no prefetching and no compression, and with no client cache is 47.6 seconds. The wall-clock time required to execute Word across a 1 Mbps link with prefetching based on profile data collected from a previous run enabled, compression enabled, and a completely empty client cache is 19.4 seconds. This greater than 50% reduction in execution time illustrates the gain due to fetching accurately predicted pages in advance along with compression of the set of pages together. And finally, the wall-clock time required to execute Word across a 1 Mbps link with prefetching based on profile data collected from a previous run enabled, compression enabled, and a cache warmed by a previous run is 4.0 seconds. This additional improvement shows that persistent caching of application file pages brings performance very close to native on subsequent runs, with minimal network load. [The latter two runs include a compression strategy reducing the bits transferred by about 40%.]

Through the use of security certificates and randomly generated keys, the invention provides a high level of security and confidence across public networks. The randomly generated key also reduces the amount of computation required to encrypt and decrypt application data while maintaining sufficient security to operate across an open network. To provide additional security to the application provider, the application stream set can be built with randomly positioned *land-mine sectors* that are associated with the application but would never be referenced during normal execution. If a *cracker* were to wrest control of the virtual device from the execution controller on a client and attempt to make a copy of the installed application, the client would request a land-mine sector which would alert the server that an act of software piracy was being attempted. The server then may choose to deny all requests from the client until the matter is properly investigated.

By providing instant execution of applications across a public network, the invention engenders new revenue models for software developers and new usage models for software consumers. Software developers can allow customers to demo or *test drive* their application in hopes of enticing more customers to buy the application. Software developers can charge per use of an application, based on either the number of times the

application is executed or by the amount of time actually spent executing the application. Software consumers also benefit because they can use their applications from any suitable computer system attached to the network. Traditional software delivery models make this very inconvenient because the consumer must carry with them the physical media containing the application and must often go through the process of un-installing the application to abide by the application's *software license agreement*.

Prior Art

US Patent 5,790,753: System for downloading computer software programs
US Patent 5,781,226: Network virtual memory for a cable television settop terminal

Uninstalling a subscribed application –

The user uses the Client UI to uninstall a subscribed application. First a check needs to be made that the application can be uninstalled. Certain applications may not be uninstallable as a unit if the subscription model does not allow that. For instance, if the ASP is selling office as a whole we may choose to only allow uninstallation of office and not of its individual components. The eStream Client File Mgr along with the License Subscription Mgr can make this decision. Once it has been identified that the application can be uninstalled, the actions outlined below must be undertaken. Note that we can implicitly decide to unsubscribe the application that is being uninstalled. However if the user only wants to uninstall an application on one of his clients but use the same application from other clients (for whatever reason) then implicitly unsubscribing would be a bad thing.

Also we need an eStream Client Mgr that manages the interaction between all the other components. All requests can go to this Manager and that can start the appropriate actions on the other components.

Actions –

- Restore the registry to the state before application installed. *This implies that we need to keep track of the entries that are modified as a result of the installation and the original values. Note that if some other application that was installed also modified the same registry entries, the values after the uninstall are not the ones at the time of the installation, rather they should be restored to what the value would have been had only the subsequent apps been installed.*
    - Components affected – Registry spoofer, App Ref/Reg Info, App Spoof Info.
    - APIs –
        - Registry spoofer – regSpoofUninstall(AppID) called from eStream Client Mgr;
        - App Ref/Reg Info – appRegInfoUninstall(AppID) called from eStream Client Mgr;
        - App Spoof Info – appSpoofUninstall(AppID) called from eStream Client Mgr;

- Remove the components in the file system specifically kept for the application. This includes persistent application code, system files that may have been modified, cache components. *This again has the same issues as the restoring of the registry in terms of the original state and the state as a result of subsequent installed applications.*
    - Components affected – File spoofer, App Ref/Reg Info, Persistent App code, Cache Mgr, eStream Client File Mgr.
    - APIs –
        - EStream Client File Mgr – appUninstall(AppID) called from eStream Client Mgr;

        - File spoofer – fileSpoofUninstall(AppID) called from eStream Client File Mgr;
        - App Ref/Reg Info – appFileInfoUninstall(AppID) called from eStream Client File Mgr;
        - Persistent App code – appRemovePersistentCode(AppID) called from eStream Client File Mgr;
        - Cache Mgr – appInvalidate(AppID) called from eStream Client File Mgr;

- Remove application short cut from start menu. *This should be easy and straightforward.*
    - Components affected – App Install Mgr.
    - APIs –
        - App Install Mgr – appRemoveShortCut(AppID) called from eStream Client Mgr;

- Remove icon that shows application when connection not established. *This should be easy and straightforward.*
    - Components affected – App Install Mgr.
    - APIs –
        - App Install Mgr – appRemoveIcon(AppID) called from eStream Client Mgr;

- Update the subscription license information if we decide to implicitly unsubscribe the application as well. *This could include sending messages to the account and the DRM servers so that the billing can stop and future accesses may be prevented. In any case we must release the current licenses being held.*
    - Components affected – License Subscription Mgr.
    - APIs –
        - License Subscription Mgr – appUnsubscribe(AppID) called from eStream Client Mgr;
        - License Subscription Mgr – appReleaseLicenses(AppID) called from eStream Client Mgr;

Uninstalling eStream client components –

The user uses the Client UI to uninstall eStream. First a check needs to be made that all the applications are uninstalled. The mechanism described above can be used to uninstall all the applications being eStreamed. The eStream Client Mgr (or some other component) must have a list of all the current installed applications and that can be used to uninstall the applications. Once the applications are uninstalled, the client components can be uninstalled.

Actions –

- For all applications that are currently installed, uninstall the applications.

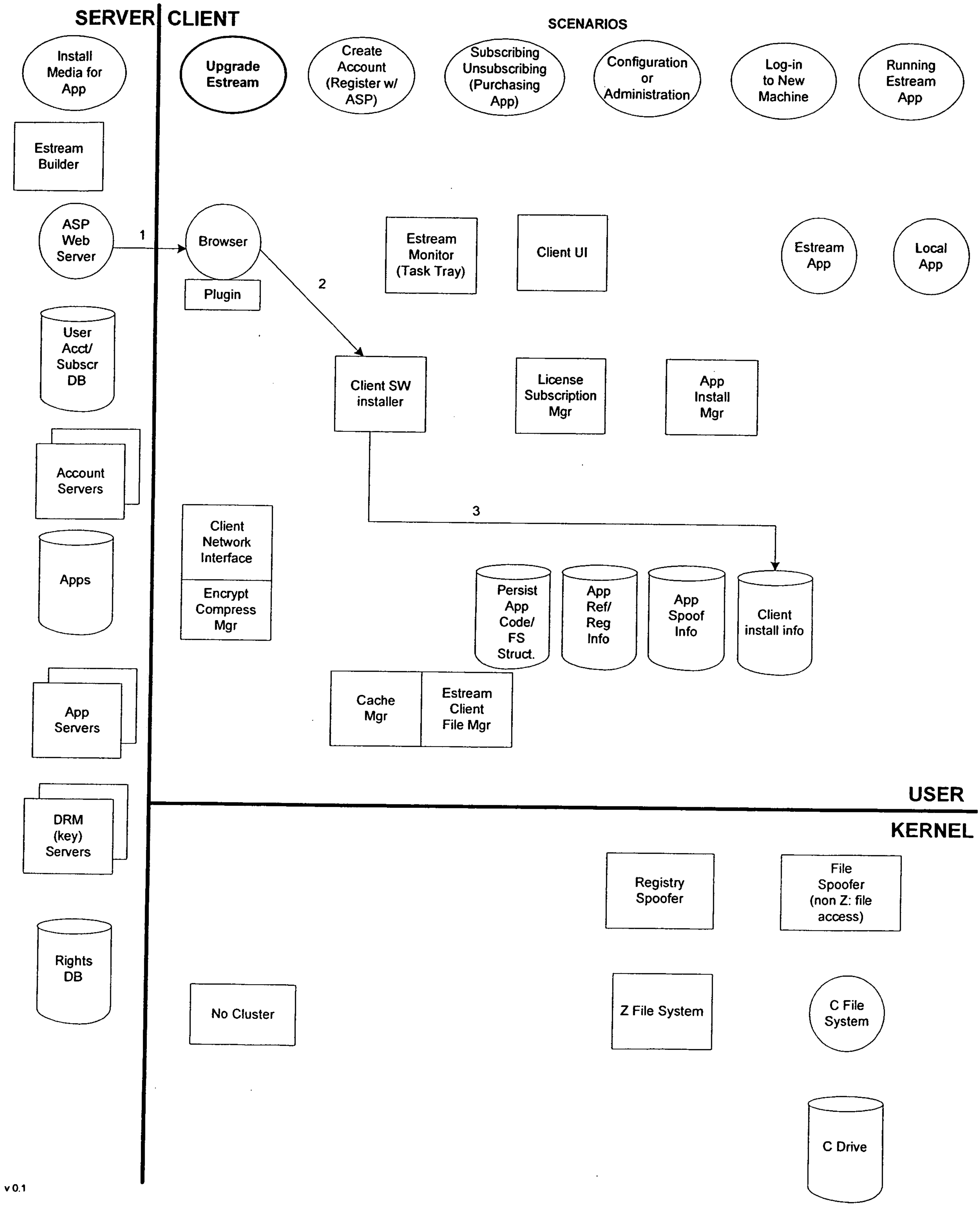
- Components – eStream Client Mgr
- APIs –
  - EStream Client Mgr – appUninstall(AppID) called from the same component

- Delete the client cache. Since all the applications are uninstalled, no one should be using the cache.
  - Components – Cache Mgr
  - APIs –
    - Cache Mgr – deleteCache(); called from the eStream Client Mgr.

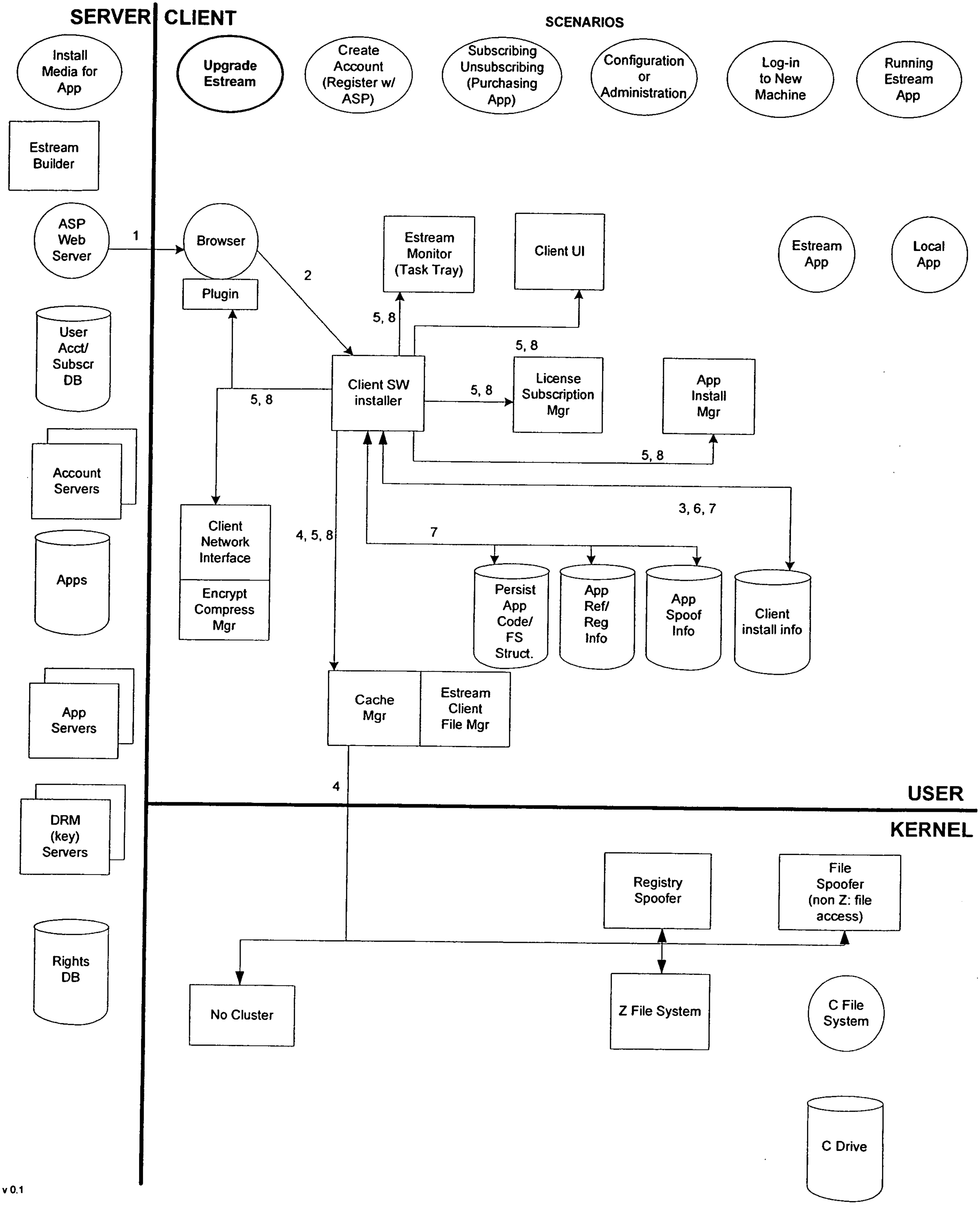
- Delete the eStream file system components. *Note that the NoCluster workaround will not disappear till the machine is rebooted but that may be reasonable.*
  - Components – eStream Client File Mgr
  - APIs –
    - EStream Client File Mgr – uninstall(); called from the eStream Client Mgr.

- Delete the eStream user level components. Use InstallShield to uninstall these.

Scenario 2: Attempt Upgrade to Older Version

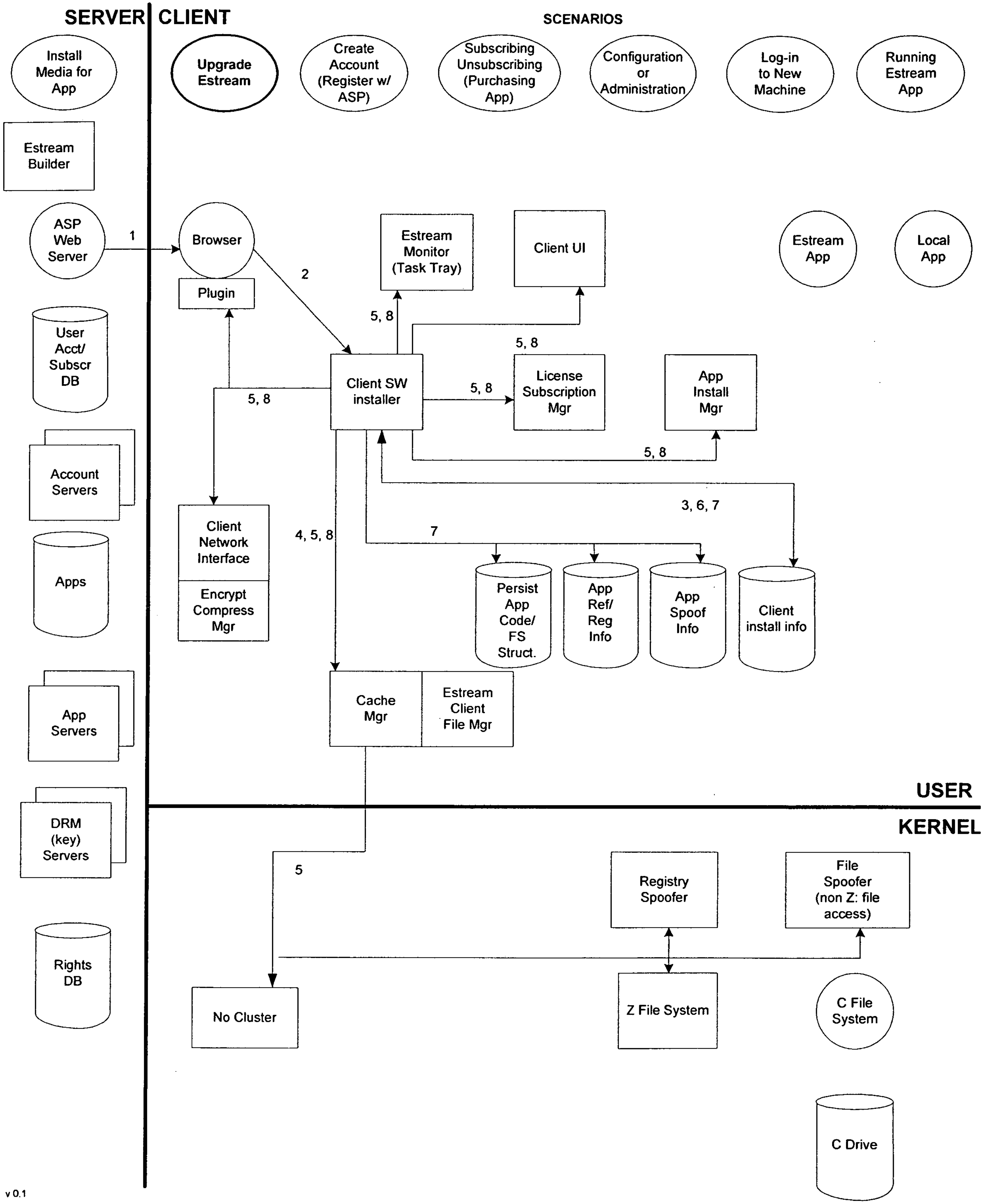


Scenario 3: User-Mode Component Upgrade

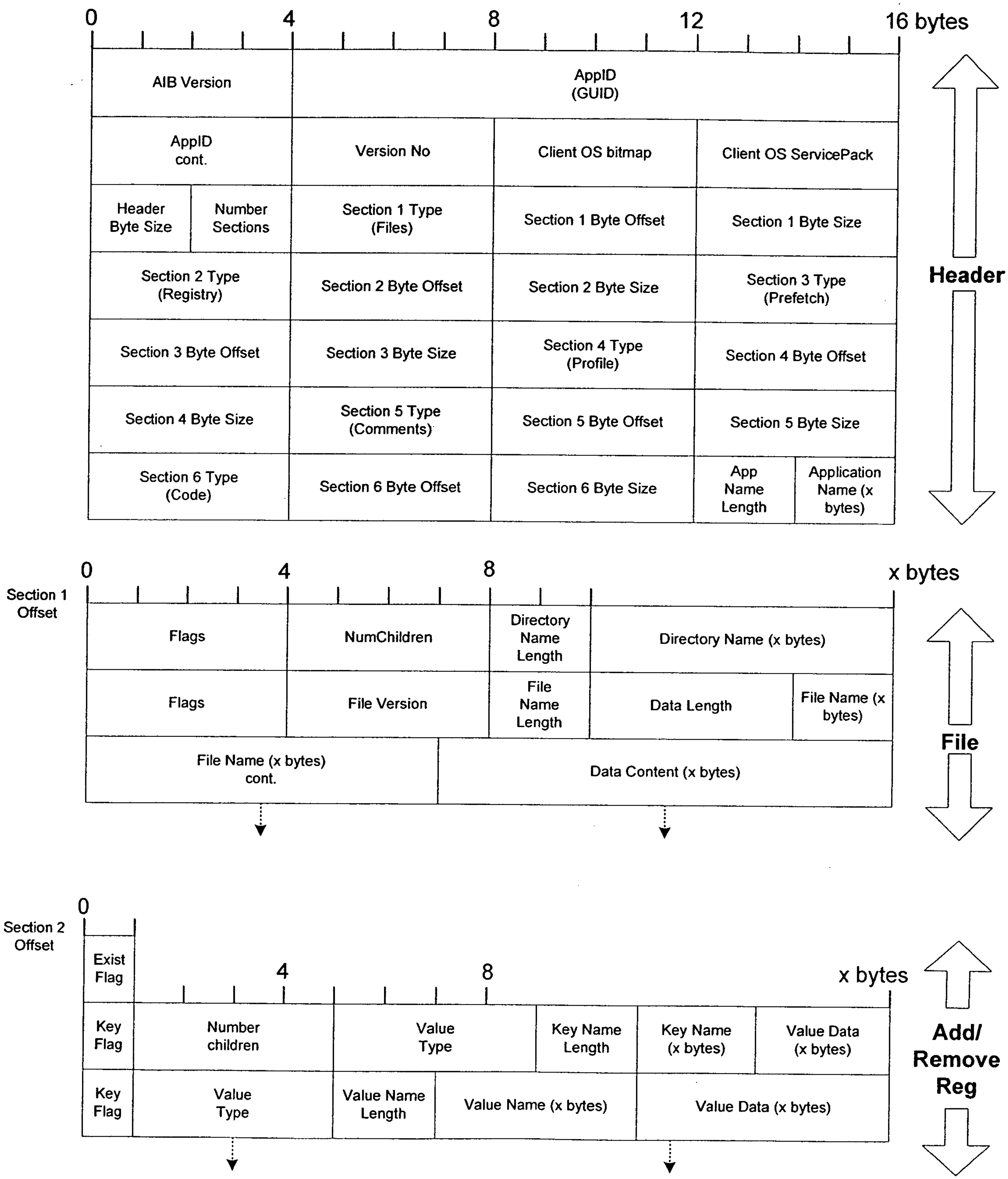


v 0.1

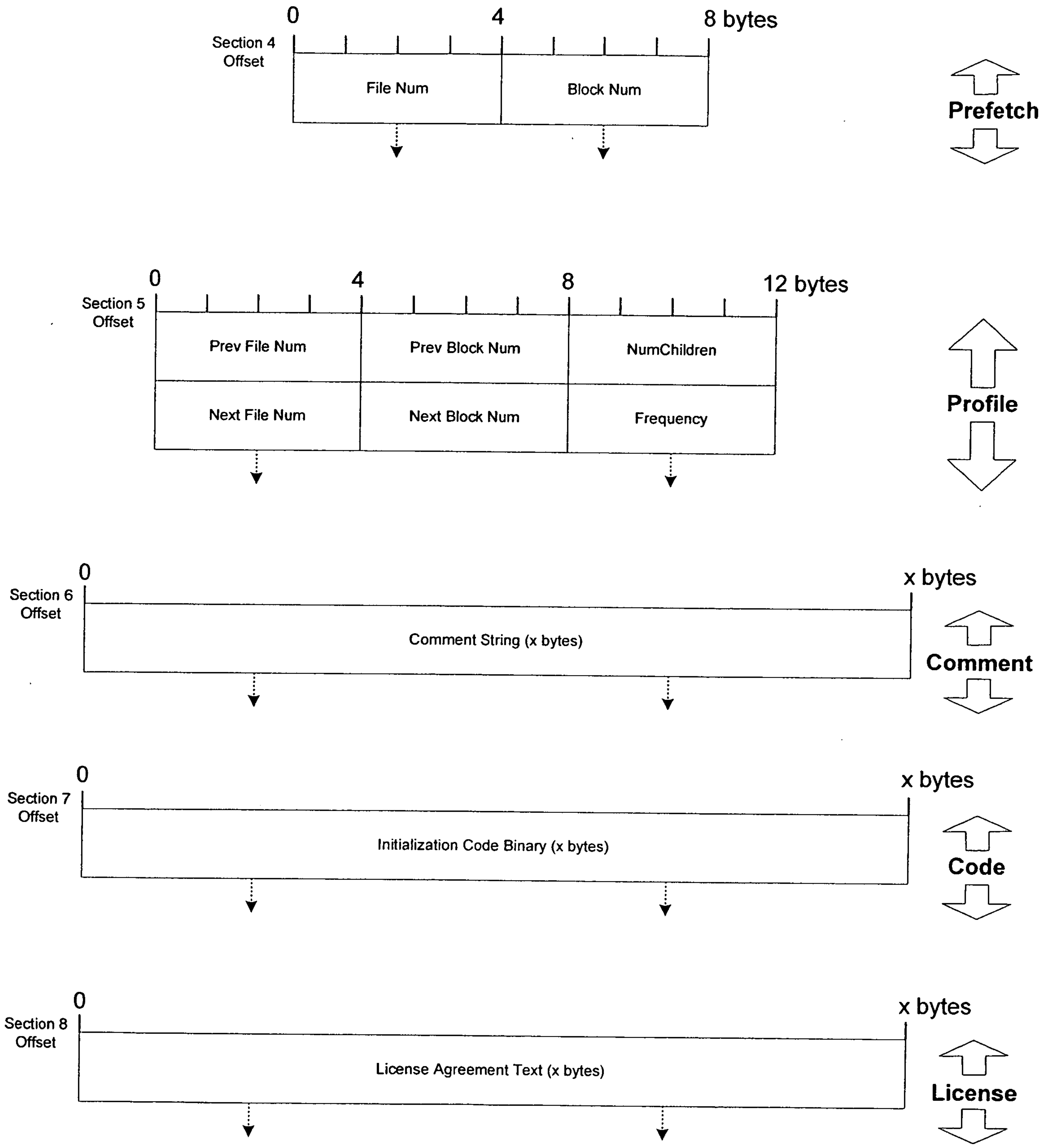
Scenario 4: User-Mode and Kernel-Mode Component Upgrade



Format of AppInstallBlock (part 1 of 2)



Format of AppInstallBlock (part 2 of 2)



eStream AppInstallBlock Low Level Design

Sanjay Pujare and David Lin
Version 0.2
[redacted]

Functionality

The AppInstallBlock is a block of code and data associated with a particular application. This AppInstallBlock contains the information needed to by the eStream client to 'initialize' the client machine before the eStream application is used for the first time. It also contains optional profiling data for increasing the runtime performance of that eStream application.

The AppInstallBlock is created offline by the eStream Builder program. First of all, the Builder monitors the installation process of a local version of the application installation program and records changes to the system. This includes any environment variables added or removed from the system, and any files added or modified in the system directories. Files added to the application specific directory is not recorded in the AppInstallBlock to reduce the amount of time needed to send the AppInstallBlock to the eStream client. Secondly, the Builder profiles the application to obtain the list of critical pages needed to run the application initially and an initial page reference sequence of the pages accessed during a sample run of the application. The AppInstallBlock contains an optional application-specific initialization code. This code is needed when the default initialization procedure is insufficient to setup the local machine environment for that particular application.

The AppInstallBlock and the runtime data are packaged into the eStream Set by the Builder and then uploaded to the application server. After the eStream client subscribed to an application and before the application is run for the first time, the AppInstallBlock is send by the server to the client. The eStream client invokes the default initialization procedure and the optional application-specific initialization code. Together, the default and the application-specific initialization procedure process the data in the AppInstallBlock to make the machine ready for eStreaming that particular application.

Data type definitions

The AppInstallBlock is divided into the following sections: header section, variable section, file section, profile section, prefetch section, comment section, and code section. The header section contains general information about the AppInstallBlock. The information includes the total byte size and an index table containing size and offset into other sections. In Windows version, the variable section consists of two registry tree structures to specify the registry entries added or removed from the OS environment. The file section is a tree structure consisting of the files copied to C drive during the application installation. The profile section contains the initial set of block reference sequences during

Builder profiling of the application. The prefetch section consists of a subset of profiled blocks used by the Builder as a hint to the eStream client to prefetch initially. The comment section is used to inform the eStream client user of any relevant information about the application installation. Finally, the code section contains an optional program tailored for any application-specific installation not covered by the default eStream application installation procedure. In Windows version, the code section contains a Windows DLL.

Here is a detailed description of each fields of the AppInstallBlock.

Note: Little endian format is used for all the fields spanning more than 1 byte. Also, BlockNumber specifies blocks of 4K byte size.

1. Header Section:

The header section contains the basic information about that AppInstallBlock. This includes the versioning information, application identification,

Core Header Structure:

- **AibVersion [4 bytes]**: Magic number or appInstallBlock version number (which identifies the version of the appInstallBlock structure rather than the contents).
- **AppId [16 bytes]**: this is an application identifier unique for each application. On Windows, this identifier is the GUID generated from the 'guidgen' program. AppId for Word on Win98 will be different from Word on WinNT if it turns out that Word binaries are different between NT and 98.
- **VersionNo [4 bytes]**: Version number. This allows us to inform the client that the appInstallBlock has changed for a particular appId. This is useful for changes to the AppInstallBlock due to minor patch upgrades in the application.
- **ClientOSBitMap [4 bytes]**: Client OS supported bitmap or ID: for Win2K, Win98, WinNT and other future OSs we might support (it should be possible to say that this appInstallBlock is for more than one OS).
- **ClientOSServicePack [4 bytes]**: We might want to store the service pack level of the OS for which this appInstallBlock has been created. Note that when this field is set we cannot use multiple OS bits in the above field ClientOSBitMap.
- **Flags [4 bytes]**: Flags pertaining to AppInstallBlock
    - **Bit 0:** Reboot – If set, the eStream client needs to reboot the machine after installing the AppInstallBlock on the client machine.
    - **Bit 1:** Unicode – If set, the string characters are 2 bytes wide instead of 1 byte.
- **HeaderSize [2 bytes]**: Total size in bytes of the header section.
- **Reserved [32 bytes]**: Reserved spaces for future.

- **NumberOfSections [1 byte]**: Number of sections in the index table. This determines the number of entries in the index table structure described below:

Index Table Structure: (variable number of entries)

- **SectionType [1 bytes]**: The type of data describe in section. 0=file section, 1=variable section, 2=prefetch section, 3=profile section, 4=comment section, 5=code section.
- **SectionOffset [4 bytes]**: The offset from the beginning of the file indicates the beginning of section.
- **SectionSize [4 bytes]**: The size in bytes of section.

Variable Structure:

- **ApplicationNameLength [4 bytes]**: Byte size of the application name
- **ApplicationName [X bytes]**: Non-null terminating name of the application

2. File Section:

The file section contains a subset of the list of files needed by the application to run properly. This section does not enumerate files located in the standard application program directory. It consists of information about files copied into 'unusual' directory during the installation of an application. If the file content is small, the file is copied to the client machine. Otherwise, the file is relocated to the standard program directory suitable for streaming. The file section data is list of trees stored in a contiguous sequence of address space according to the pre-order traversal of the trees. A node in the tree can correspond to one or more levels of directory. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal Windows pathname including the drive letter. Each entry of the node in the tree consists of the following structure:

Directory Structure: (variable number of entries)

- **Flags [4 byte]**: Bit 0 is set if this entry is a directory
- **NumberOfChildren [2 bytes]**: Number of nodes in this directory
- **DirectoryNameLength [4 bytes]**: Length of the directory name
- **DirectoryName [X bytes]**: Non-null terminating directory name

Leaf Structure: (variable number of entries)

- **Flags [4 byte]**: Bit 1 is set to 1 if this entry is a spoof or copied file name
- **FileVersion [4? bytes]**: Version of the file GetFileVersionInfo() if the file is win32 file image. Need variable file version size returned by GetFileVersionInfoSize(). Otherwise use GetFileTime() to retrieve the file creation time.
- **FileNameLength [4 bytes]**: Byte size of the file name

- **DataLength [4 bytes]**: Byte size of the data. If spoof file, then data is the string of the spoof directory. If copied file, then data is the content of the file
- **FileName [X bytes]**: Non-null terminating file name
- **Data [X bytes]**: Either the spoof file name or the content of the copied file

3. Add Variable and Remove Variable Sections:

The add and remove variable sections contain the system variable changes needed to run the application. In Windows system, each section consists of several number of registry subtrees. Each tree is stored in a contiguous sequence of address space according to the pre-order traversal of the tree. A node in the tree can correspond to one or more levels of directory in the registry. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal key name. The order of the trees is shown here.

a. Registry Subsection:

1. "KHCR": HKEY_CLASSES_ROOT
2. "HKCU": HKEY_CURRENT_USER
3. "HKLM": HKEY_LOCAL_MACHINE
4. "HKU": HKEY_USERS
5. "HKCC": HKEY_CURRENT_CONFIG

Tree Structure: (5 entries)

- **ExistFlag [1 byte]**: Set to 1 if this tree exist, 0 otherwise.
- **Key or Value Structure entries [X bytes]**: Serialization of the tree into variable number key or value structures described below.

Key Structure: (variable number of entries)

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **NumberOfSubchild [4 bytes]**: Number of subkeys and values in this key directory
- **KeyNameLength [4 bytes]**: Byte size of the key name
- **KeyName [X bytes]**: Non-null terminating key name

Value Structure: (variable number of entries)

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **ValueType [4 byte]**: Type of values from the Win32 API function RegQueryValueEx(): REG_SZ, REG_BINARY, REG_DWORD, REG_LINK, REG_NONE, etc...
- **ValueNameLength [4 bytes]**: Byte size of the value name
- **ValueDataLength [4 bytes]**: Byte size of the value data

- **ValueName [X bytes]**: Non-null terminating value name
- **ValueData [X bytes]**: Value of the Data

In addition to registry changes, an installation in Windows system may involve changes to the ini files. The following structure is used to communicate the ini file changes needed to be done on the eStream client machine. The ini entries are appended to the end of the variable section after the 5 registry trees are enumerated.

b. INI Subsection:

- **NumFiles [4 bytes]**: Number of INI files modified.

File Structure: (variable number of entries)

- **FileNameLength [4 bytes]**: Byte length of the file name
- **FileName [X bytes]**: Name of the INI file
- **NumSection [4 bytes]**: Number of sections with the changes

Section Structure: (variable number of entries)

- **SectionNameLength [4 bytes]**: Byte length of the section name
- **SectionName [X bytes]**: Section name of an INI file
- **NumValues [4 bytes]**: Number of values in this section

Value Structure: (variable number of entries)

- **ValueLength [4 bytes]**: Byte length of the value data
- **ValueData [X bytes]**: Content of the value data

4. Prefetch Section:

The prefetch section contains a list of file blocks. The Builder profiler determines the set of file blocks critical for the initial run of the application. This data includes the code to start and terminate the application. It includes the file blocks containing for frequently used commands. For example, opening and saving of documents are frequently used commands and should be prefetched if possible. Another type of blocks to include in the prefetch section is the blocks associated with frequently accessed directories and file metadata in this directory. The format of the data is described below:

- **FileNumber [4 bytes]**: File Number of the file containing the block to prefetch
- **BlockNumber [4 bytes]**: Block Number of the file block to prefetch

5. Profile Section: (not used in eStream 1.0)

The profile section consists of a reference sequence of file blocks accessed by the application at runtime. Conceptually, the profile data is a two dimensional matrix. Each entry [*row, column*] of the matrix is the frequency a block *row* is followed by a block *column*. In any realistic applications of fair size, this matrix is very large and sparse. Proper data structure must be selected to store this sparse matrix efficiently in required storage space and minimize the overhead in accessing this data structure access.

The section is constructed from two basic structures: row and column structures. Each row structure is followed by N column structures specified in the NumberColumns field. Note that this is an optional section. But with appropriate profile data, the eStream client prefetcher performance can be increased.

Row Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the row block
- **BlockNumber [4 bytes]**: Block Number of the row block
- **NumberColumns [4 bytes]**: number of blocks that follows this block. This field determines the number of column structures following this field.

Column Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the column block
- **BlockNumber [4 bytes]**: Block Number of the column block
- **Frequency [4 bytes]**: frequency the row block is followed by column block

6. Comment Section:

The comment section is used by the Builder to describe this AppInstallBlock in more detail.

- **Comment [X bytes]**: Null terminating comment string

7. Code Section:

The code section consists of the application-specific initialization code needed to run on the eStream client to setup the client machine for this particular application. This section may be empty if the default initialization procedure in the eStream client is able to setup the client machine without requiring any application-specific instructions. On the Windows system, the code is a DLL file containing two exported function calls: *Install*(), *Uninstall*(). The eStream client loads the DLL and invokes the appropriate function calls.

- **Code [X bytes]**: Binary file containing the application-specific initialization code. On Windows, this is just a DLL file.

8. LicenseAgreement Section:

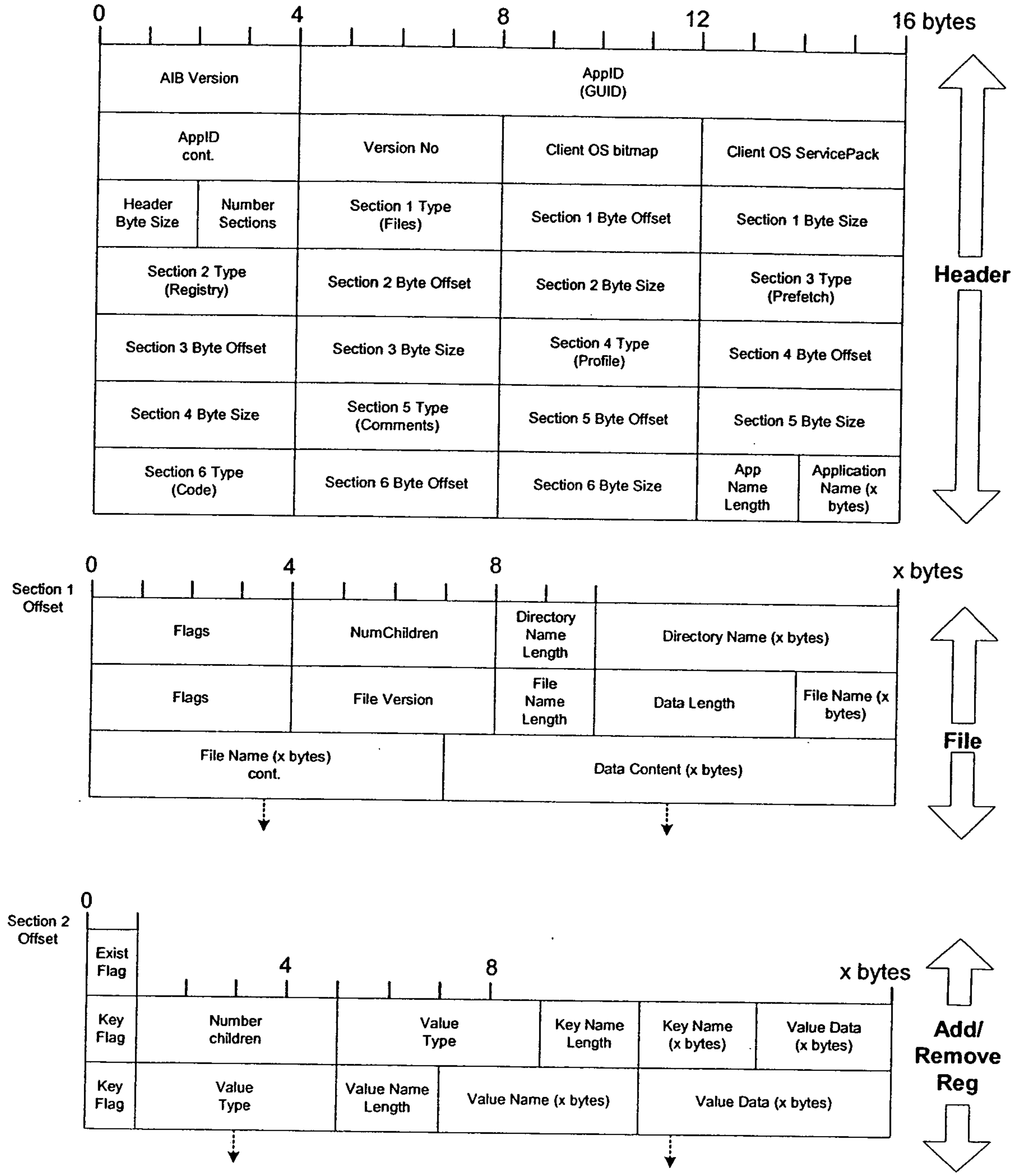
The Builder creates the license agreement section. The eStream client displays the license agreement text to the end-user before the application is started for the first time. The end-user must agree to all licensing agreement set by the software vendor in order to use the application.

- **LicenseAgreement [X bytes]**: Null terminating license agreement string

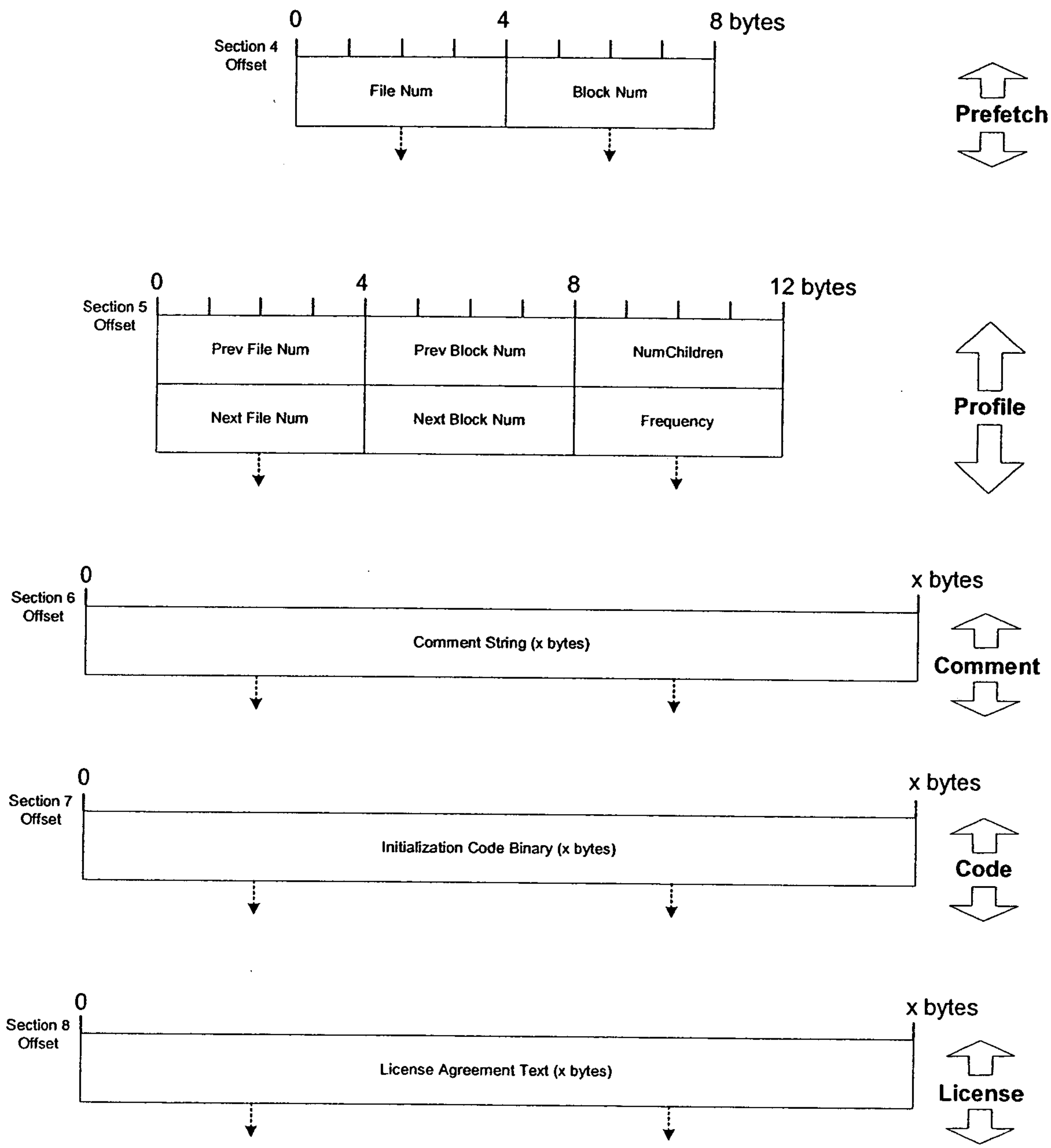
Open Issues

- What is the size of the AppInstallBlock for a typical application like Office?
- How large should the prefetch sections be for optimal run of an application? At minimum, it should contain at least start/termination code.
- How should the AppInstallBlock handle application license agreement text string? Add a new section or use comment section. Does the dialog need to have exactly the same interface as the license agreement dialog on the local installation?
- Currently, file section stores complete pathname including the drive letter. The installation may place files according to some variables like %SystemRoot% or %UserProfile%. How does the Builder detect this so it can propagate this information to the client?

Format of AppInstallBlock (part 1 of 2)



Format of AppInstallBlock (part 2 of 2)



The eStream Builder

The eStream Builder is a software program. It is used to convert locally installable applications into a data set suitable for streaming over the network. The streaming-enabled data set is called the eStream Set. This document describes the procedure used to convert locally installable applications into the eStream Set.

The application conversion procedure into the eStream Set consists of the several steps. In the first phase, the Builder program monitors the installation process of a local installation of the desired application for conversion. The Builder monitors any changes to the system and records those changes in an intermediate data structure. After the application is installed locally, the Builder enters the second phase of the conversion. In the second step, the Builder program invokes the installed application executable and obtains sequences of frequently accessed file blocks of this application. Both the Builder program and the eStream client software use the sequence data to optimize the performance of the streaming process. Once the sequencing information is obtained, the Builder enters the final phase of the conversion. In this step, the Builder gathers all data obtained from the first two phase and processes the data into the eStream Set.

In the next sections, detailed descriptions of the three phases of the Builder conversion process are described. The three phases consists of installation monitoring, application profiling, and finally eStream packaging. In most cases, the conversion process is general and applicable to all type of system. In places where the conversion is OS dependent, the discussion is focused on Microsoft Windows environment. Issues on conversion procedure for other OS environment are described in later sections.

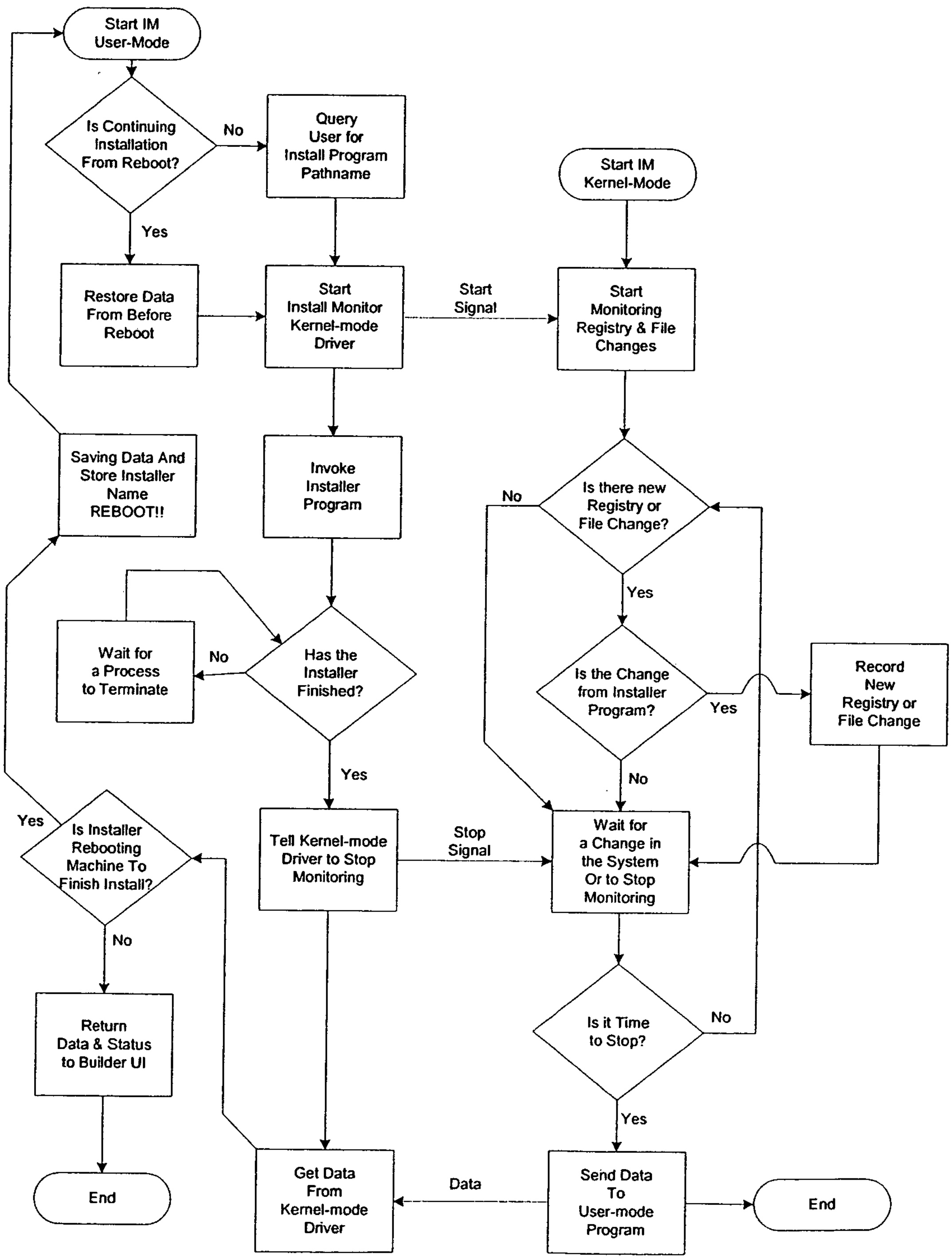
Installation Monitoring
In the first phase of the conversion process, the Builder Installation Monitor (IM) component invokes the application installation program that installs the application locally. The IM observes all changes to the local computer during the installation. The changes may involve one or more of the following: changes to system or environment variables; and modifications, addition, or deletion of one or more files. The IM records all changes to the variables and files in a data structure to be sent to the Builder's eStream Packaging component. In the following paragraphs, detailed description of the Installation Monitor is described for Microsoft Windows environment.

In Microsoft Windows system, the Installation Monitor (IM) component consists of a kernel-mode driver subcomponent and a user-mode subcomponent. The kernel-mode driver is hooked into the Windows registry and file system function interface calls. The hook into the registry function calls allows the IM to monitor system variable changes. The hook into the file system function calls enables the IM to observe file changes.

The IM kernel-mode (IM-KM) driver subcomponent is controlled by the user-mode subcomponent (IM-UM). The IM-UM sends messages to the IM-KM to start and stop the monitoring process via standard I/O control messages called IOCTL. The IM-KM memorizes any addition or deletion of registry variables. It also records changes to

application-specific, shared among a group of applications, or system-wide files. Every files and directories are assigned a unique file number for simplifying identification of a specific file. Once the installation of an application completed, the IM-UM retrieves these changes from the IM-KM and forward the data structure to the eStream Packager.

Builder Install Monitor Control Flow Diagram



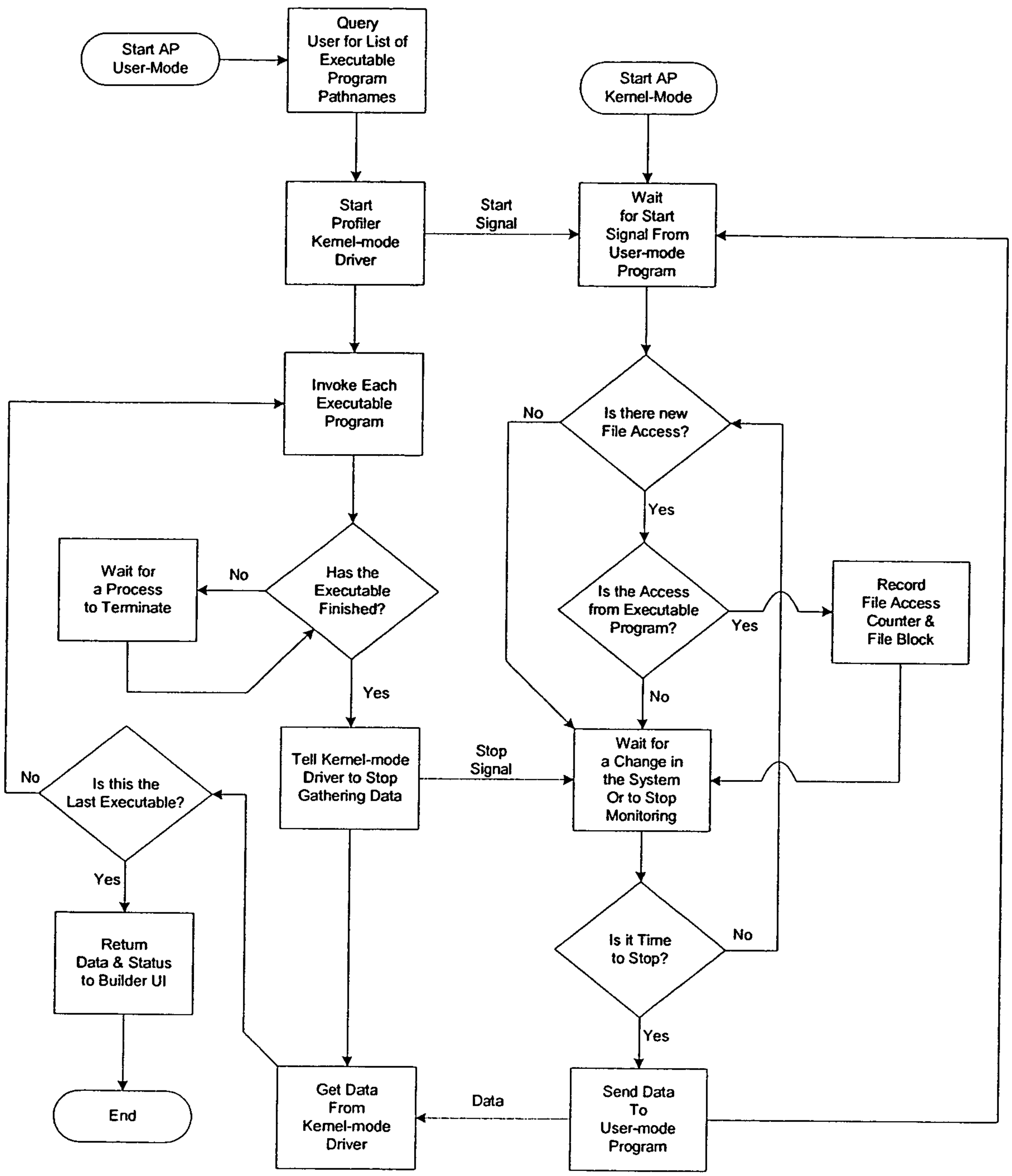
Application Profiling
In the second phase of the conversion process, the Builder's Application Profiler (AP) component invokes the application executable program that is installed during the first phase of the conversion process. The executable program files are accessed in a particular sequence. And the purpose of the AP is to captures this sequence data. This data is useful in several ways.

First of all, frequently used file blocks can be streamed to the eStream client before other less used file blocks. A frequently used file blocks is cached locally on the eStream client cache before the user starts using the streamed application for the first time. This has the effect of making the streamed application as responsive to the user as the locally installed application by hiding any long network latency and bandwidth problems.

Secondly, the frequently accessed files can be reordered in the directory to allow faster lookup. This optimization is useful for directories with large number of files. When the eStream client looks up a frequently used file in a directory, it finds this file early in the directory search. In an application run with many directory queries, the potential performance gain is significant.

The Application Profiler (AP) is not as tied to the system as the Installation Monitor (IM) but there is still some OS dependent issue. In the Windows system, the AP still has two subcomponents: kernel-mode (AP-KM) subcomponent and the user-mode (AP-UM) subcomponent. The AP-UM invokes the converting application executable. Then AP-UM starts the AP-KM to track the sequences of file block accesses by the application. Finally when the application exits after the desired amount of sequence data is gathered, the AP-UM retrieves the data from AP-KM and forwards the data to the eStream Packager.

Builder Profiler Control Flow Diagram



EStream Packaging

In the final phase of the conversion process, the Builder's eStream Packager (EP) component processes the data structure from IM and AP to create a data set suitable for streaming over the network. This converted data set is called the eStream Set and is suitable for uploading to the eStream Servers.

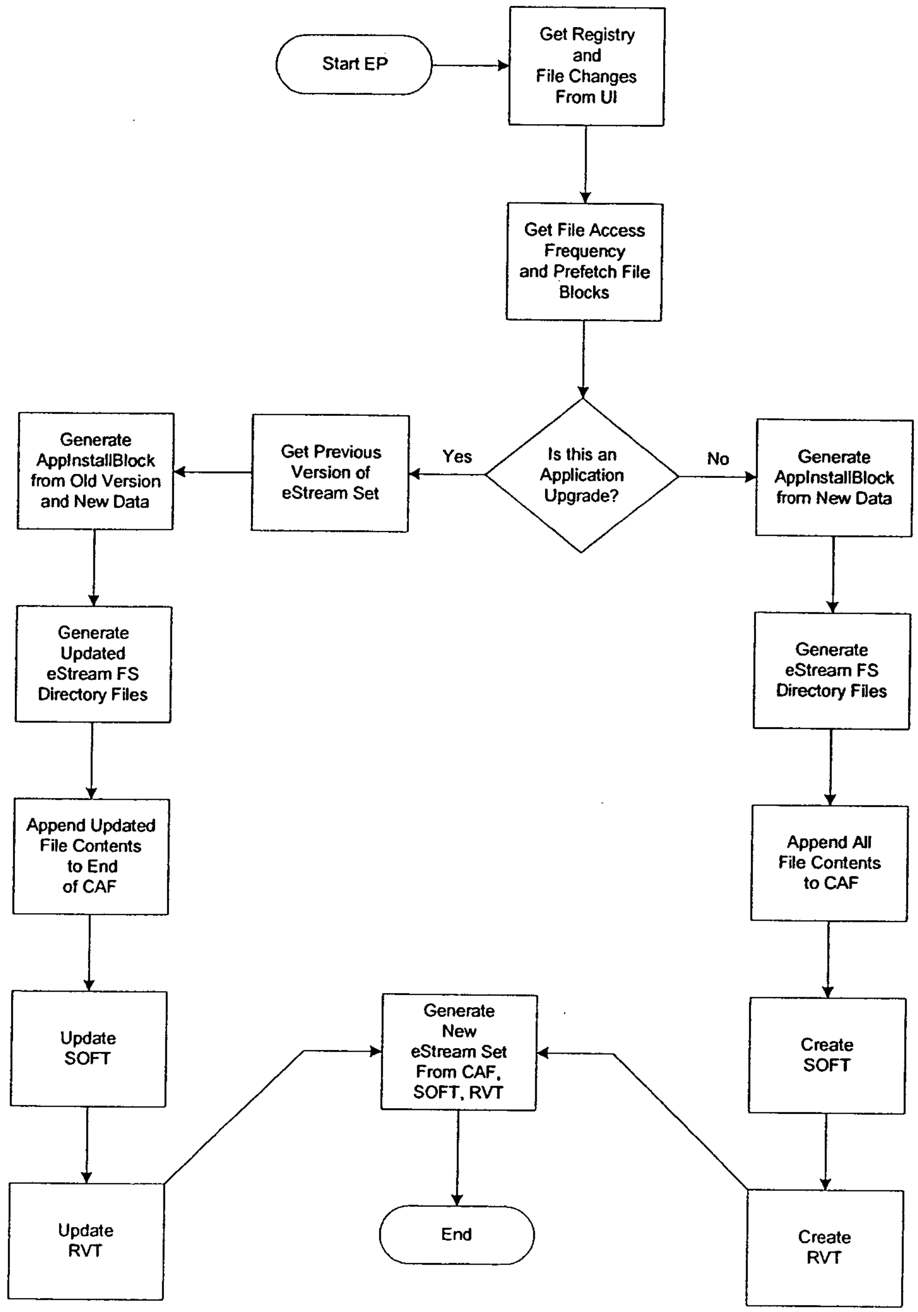
The eStream Set consists of the three sets of data from the eStream Server's perspective. The three types of data are Concatenation Application File (CAF), Size Offset File Table (SOFT), and Root Versioning Table (RVT).

The Concatenation Application File (CAF) consists of all the files and directories needed to stream to the client. The CAF can be further divided into two subsets: initialization data set and the runtime data set. The initialization data set is the first set of data to be streamed from the server to the client. This data set contains the information captured by IM and AP needed by the client to prepare the client machine for eStreaming this particular application. This initialization data set is also called the AppInstallBlock. Detailed format description of the AppInstallBlock is described in another document. The second part of the CAF consists of the runtime data set. This is the rest of the data that is streamed to the client once the client machine is initialized for this particular application. The EP appends every files recorded by IM into the CAF and generates all directories. Each directory contains list of file name, file number, and the metadata associated with the files in that particular directory.

The EP is also responsible for generating the SOFT file. This is a table used to index into the CAF for determining the start and the end of a file. The server uses this information to quickly access the proper file within the directory.

Finally, the EP creates the RVT file. The Root Versioning Table contains a list of root file number and version number. This information is used to track minor application patches and upgrades. The EP generates new directories when any single file is changed from the patch upgrade. The RVT is uploaded to the server and requested by the eStream client at appropriate time for the most updated version of the application by a simple comparison of the client's eStream application root file number with the RVT table located on the server.

Builder eStream Packager Control Flow Diagram



Data Flow Description

The following list describes the data that is passed from one component to another. The numbers corresponds to the numbering in the Data Flow diagram.

1. The full pathname of the installer program is query from the user of the Builder program and is sent to the Install Monitor.

2. The Install Monitor (IM) user-mode sends a read request to the hard-drive controller to spawn a new process for installing the application on the local machine.
3. The OS loads the application installer program into memory and run the installer program.
4. The installer program reads more files from the CD media.
5. The CD media data files are read into memory by the installer program.
6. The application installer program writes the files into proper locations on the local hard-drive.
7. IM kernel-mode captures all file read/write requests and all registry read/write requests by the installer program.
8. IM kernel-mode program sends the list of all file changes and all registry changes to the IM user-mode program.
9. IM user-mode identify special files which needs to be copied or spoofed into eStream client machine before the regular files can be streamed. It also assigns unique file numbers to every file. This data is returned to the Builder UI.
10. Builder UI invokes Application Profiling (AP) user-mode program by querying the user for the list of application executable names to be profiled.
11. Application Profiler user-mode invokes each application executable in succession by spawning each program in a new process.
12. The OS loads the application executable into memory and run the executable.
13. The executable file image is loaded into memory and starts executing. The application files will continuously be loaded into memory as needed.
14. Every file accesses to load the application file blocks into memory is monitored by the Application Profiler (AP) kernel-mode.
15. Application Profiler kernel-mode returns the file access sequence and frequency information to the user-mode program.
16. Application Profiler returns the processed profile information. This has two sections. The first section is used to identify frequency of files accessed. The second section is used to list the file blocks for prefetch to the client.
17. The eStream Packager receives files and registry changes from the Builder UI. It also receives the file access frequency and a list of file blocks from the Profiler.
18. The eStream Packager reads all file data from the hard-drive that are copied there by the application installer.
19. The eStream Packager also reads the previous version of eStream Set for support of minor patch upgrades.
20. Finally, the new eStream Set data is stored back to non-volatile storage.

Mapping of Data Flow to eStream Set

Step 7: Data gathered from this step consist of the registry and file changes. This data is mapped to the AppInstallBlock's File Section, Add Registry Section, and Remove Registry Section.

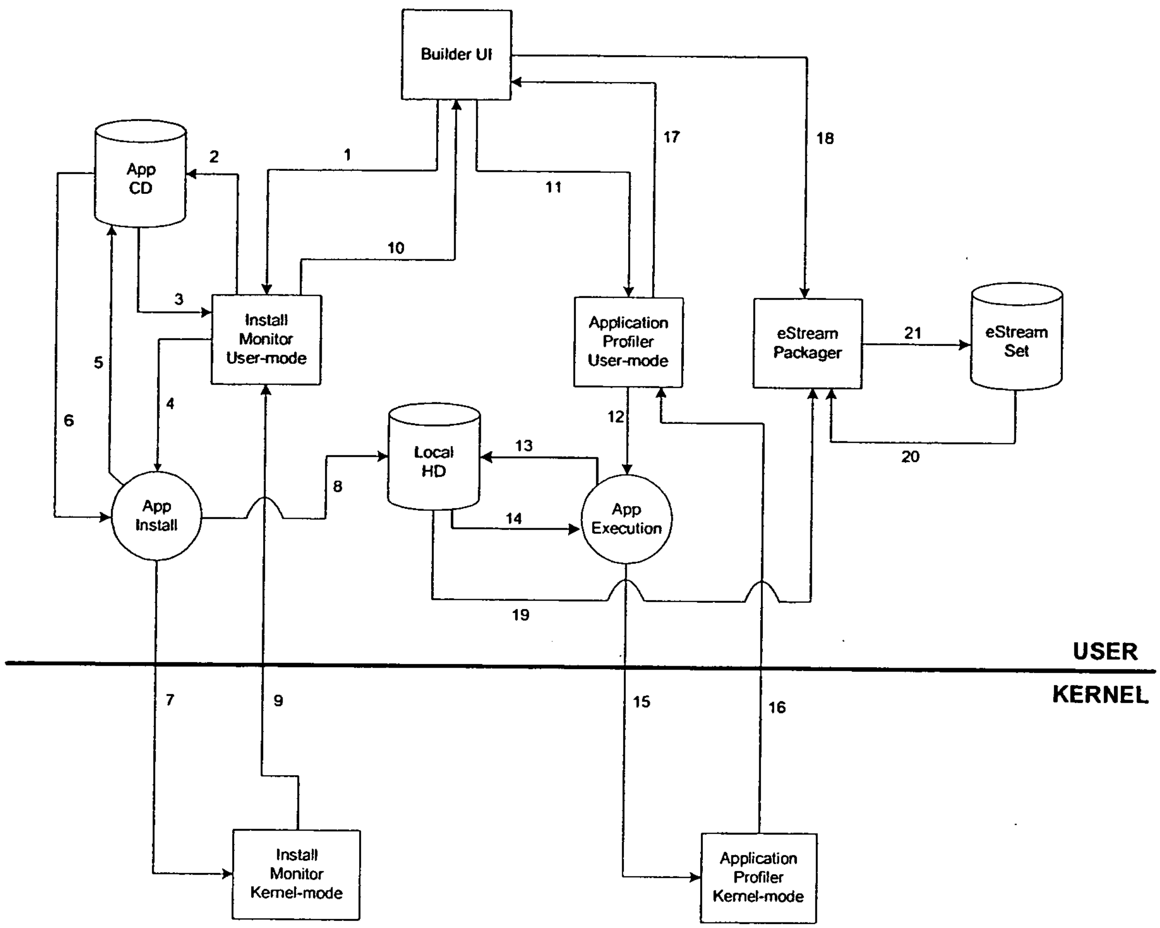
Step 8 & 19: File data are copied to the local hard-drive then concatenated into part of the CAF contents.

Step 10: Data returned to the Builder UI contains unique file numbers. This data is mapped to the file numbers used throughout the eStream Set data structure.

Step 15: Part of the data gathered by the Profiler is used to generate a more efficient eStream FS Directory content. Another part of the data is used in the AppInstallBlock as a prefetch hint to the eStream client.

Step 20: If the installation program was an upgrade, eStream Packager needs previous version of the eStream Set data. Appropriate data from the previous version is combined with the new data to form the new eStream Set.

eStream Builder Data Flow Diagram

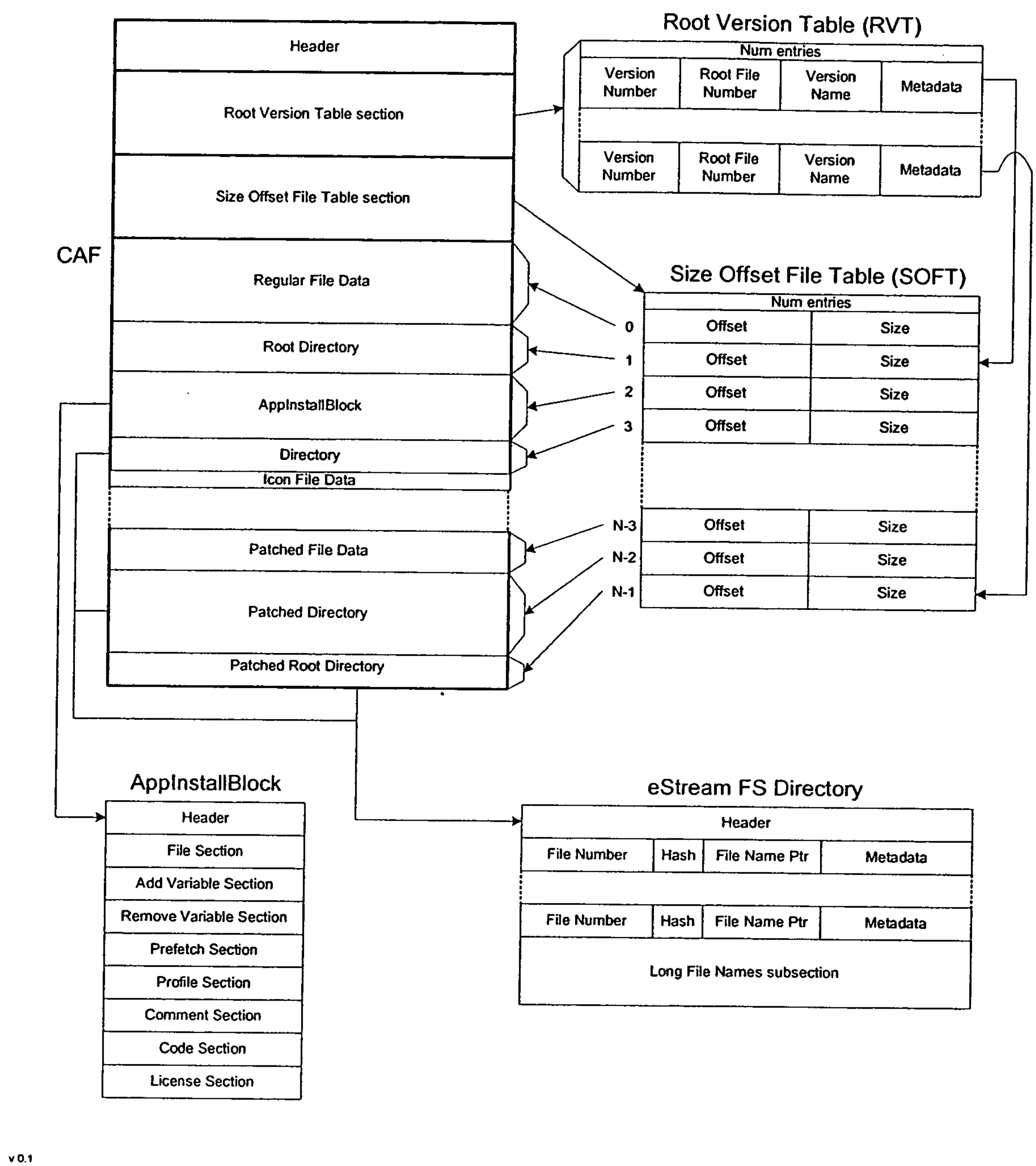


Format of eStream Set
The format of the eStream Set consists of 3 sections: Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF). The RVT section lists all versions of the root file numbers available in an eStream Set. The SOFT section consists of the pointers into the CAF section for every file in the CAF. The CAF section contains the concatenation of all the files. The CAF section is made up of regular application files, eStream FS directory files, AppInstallBlock, and icon files. Please see the document on eStream Set Format for detailed format of the eStream Set.

OS dependent format
The format of the eStream Set is designed to be as portable as possible across all OS platforms. At the highest level, the format of CAF, SOFT, and RVT that make up the format of eStream Set are completely portable across any OS platforms. The only critical piece of data structure that is OS dependent is located in the initialization data set called AppInstallBlock in the CAF. This data is dependent on the type of OS due to the differences in low-level system differences among different OS. For example, the Microsoft Windows contain system environment variables called the Registry. The Registry has a particular tree format not found in other operating systems like UNIX or MacOS.

Another OS dependent format is the format of the file names. Applications running on the Windows environment inherit the old MSDOS 8.3 file name format. To support this properly, the format of the Directory file in CAF requires an additional 8.3 field. This field is not needed in other operating systems like UNIX or MacOS.

Format of the eStream Set



Device driver versus file system paradigm

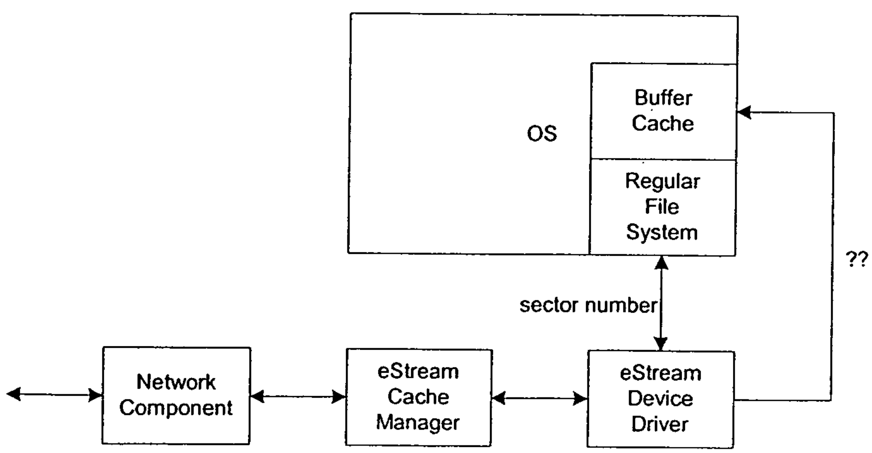
The eStream Prototype is implemented using the 'device driver' paradigm. One of the advantages of the device driver approach is that the caching of the sector blocks is

simpler. The client cache manager only needs to track sector number in its cache. In comparison with the 'file system' paradigm, more complex data structure is required to track a subset of a file that is cached on a client machine. This makes 'device driver' paradigm easier to implement.

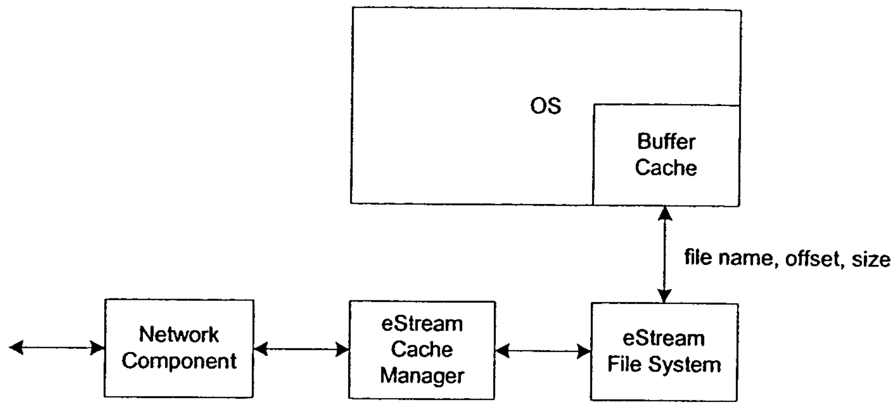
On the other hand, there are many drawbacks to the 'device driver' paradigm. On the Windows system, the device driver approach has problem supporting large number of applications. This is due to the limitation on the number of assignable drive letters available in a Windows system (26 letters); and the fact that each application needs to be located in its own device. Note that having multiple applications in a device is possible, but then the server needs to maintain exponential number of devices that support all possible combinations of applications. This is too costly to maintain on the server.

Another problem with the device driver approach is that the device driver operates at the disk sector level. This is a much lower level than operating at the file level in the file system approach. The device driver does not know anything about files. Thus, the device driver cannot easily interact with the file level issues. For example, spoofing files and interacting with OS buffer cache is nearly impossible with device driver approach. But both spoofing files and interacting with OS buffer cache is need to get higher performance.

Device Driver Paradigm



File System Paradigm



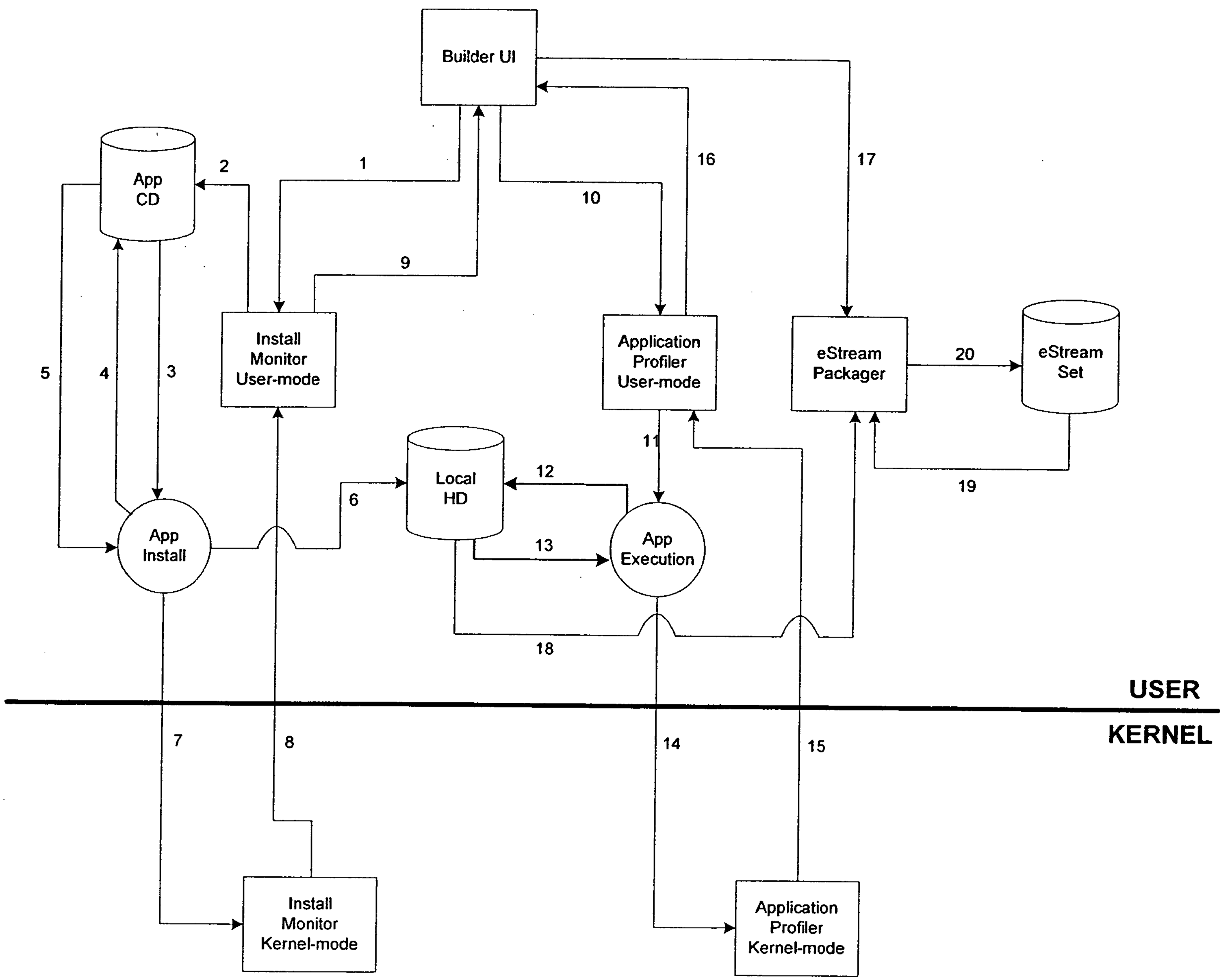
Implementation in the Prototype
The prototype has been implemented and tested successfully on the Windows and Linux distributed system. The prototype is implemented using the 'device driver' paradigm as described above. The exact procedure for streaming application data is described next.

First of all, the prototype server is started on either the Windows or Linux system. The server creates a large local file mimicking large local disk images. Once the disk images are prepared, it listens to TCP/IP ports for any disk sector read or write requests.

Secondly, the conversion process is done on a Windows system via semi-manual procedure. The server disk image is 'mounted' on the local Z drive by making the proper TCP/IP connection to the server. Then the application installation program is invoked and the application is installed into the Z drive. This writes the application files into the Z drive device driver, through the TCP/IP connection, and finally on to the server disk image. At the same time, a file and registry monitoring program records all registry and file changes. This data is stored as an initialization file to be invoked on the client to prepare the client machine for streaming.

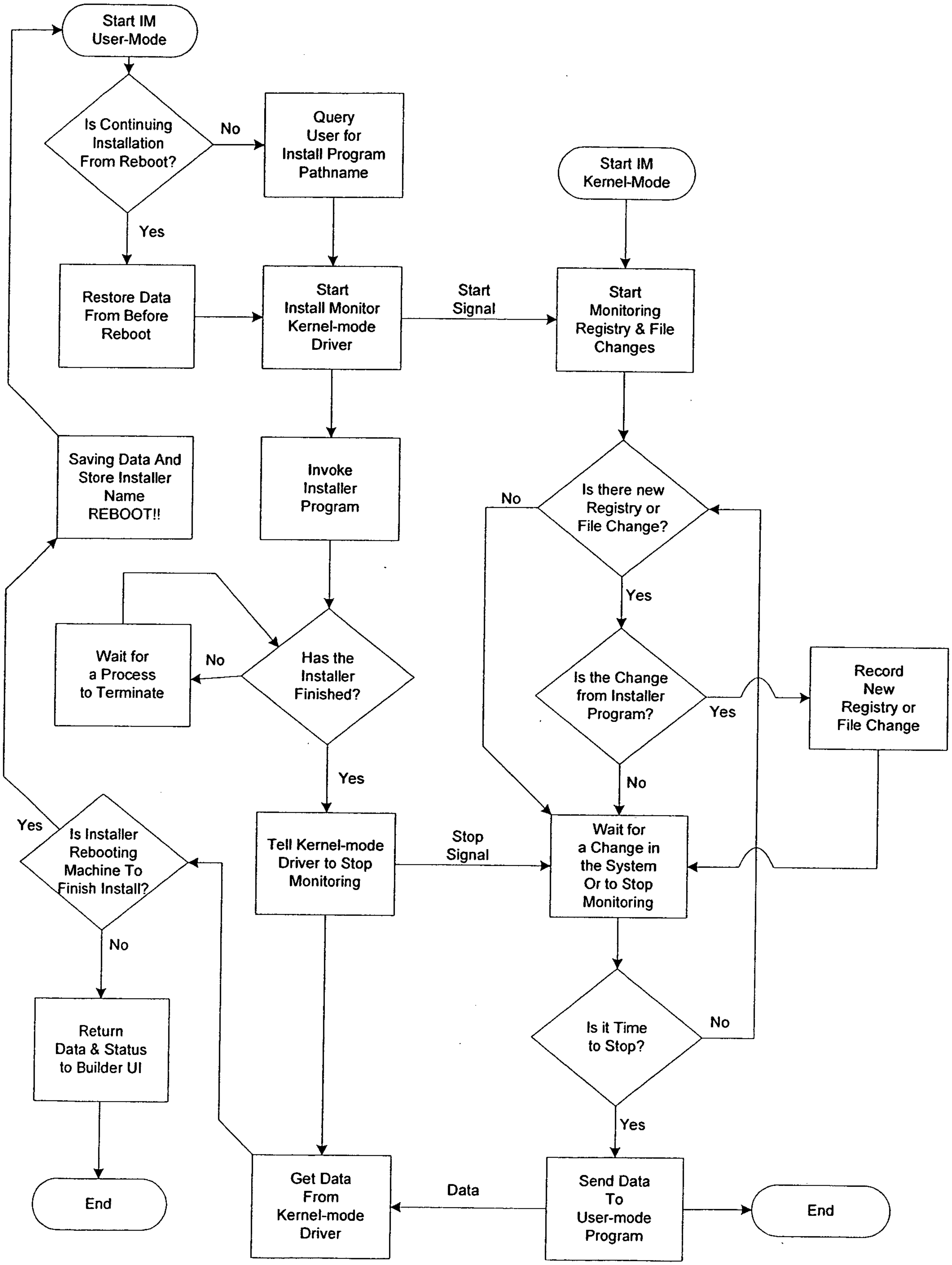
Finally, after the application files is stored on the server disk image, the client prototype is started. The client connects to the server and 'mount' the server disk image as a local Z drive. Then the initialization file is invoked which setup the local registry variables and copy system files into proper directories. Once the local machine is prepared for streaming that particular application, the user can start using the application. When the application is first started, the pages are not located in the local buffer cache. The OS makes sector request to the eStream device driver that forwards the sector request to the eStream Cache Manager. If the sector is located in the eStream cache, then the data is returned immediately. If the data is not located in the eStream cache, then the request forwarded to the network component that sends the message to the server. The server finds the proper sector data and returns the data to the client. The client eStream Cache Manager caches the new sector data and forwards the sector data to the eStream device driver. The device driver returns the sector data to the OS.

eStream Builder Data Flow Diagram

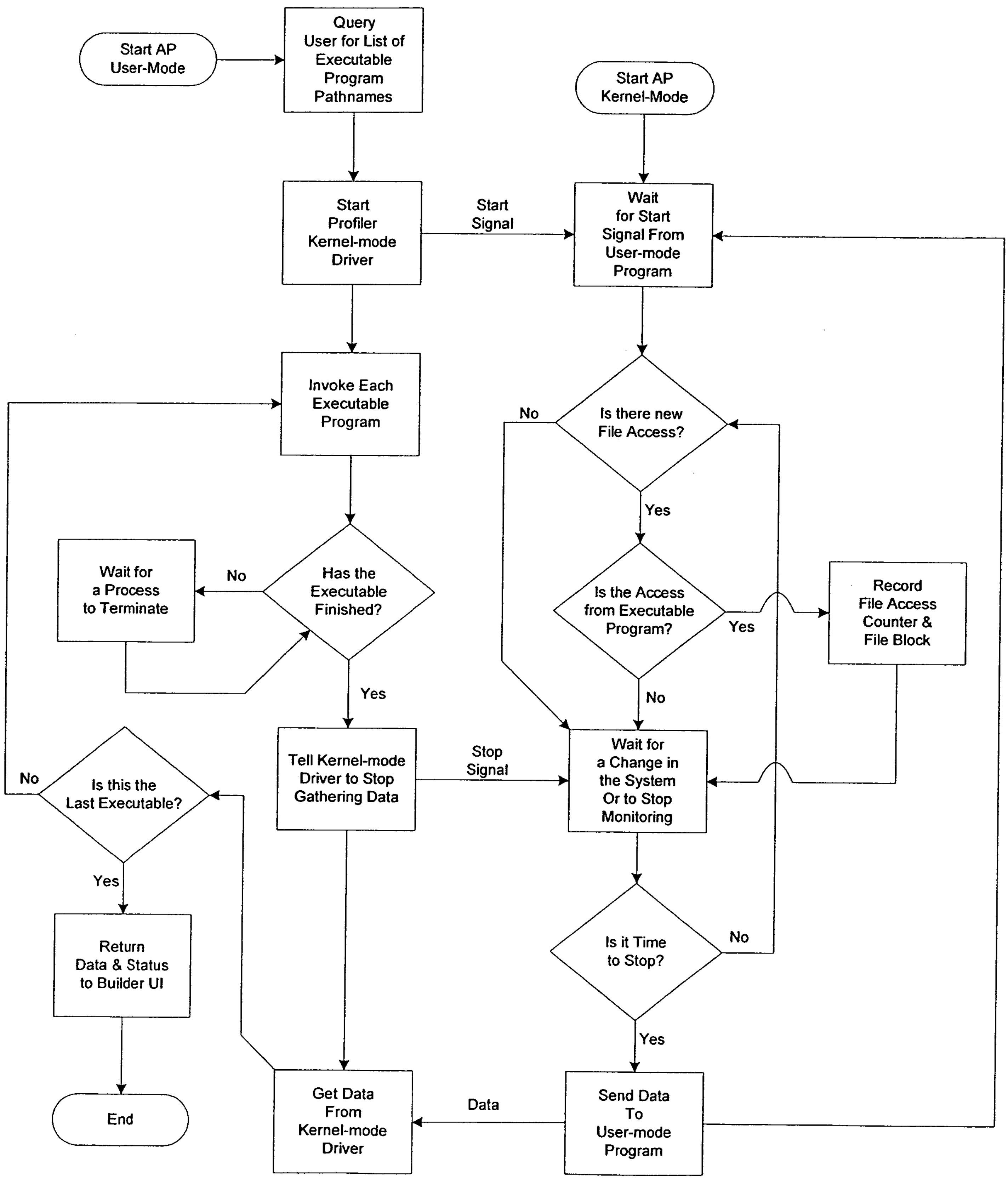


v 0.1

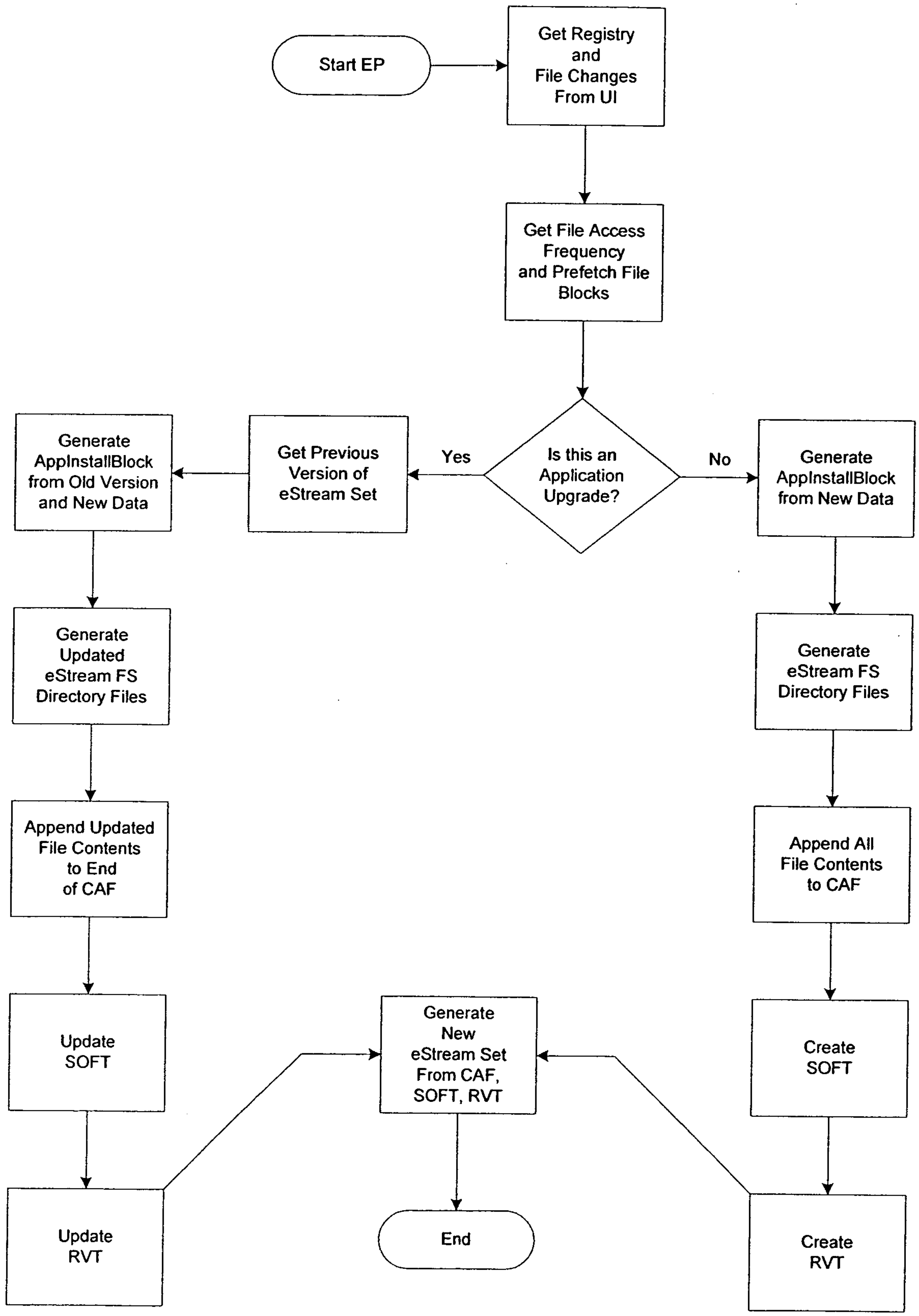
Builder Install Monitor Control Flow Diagram



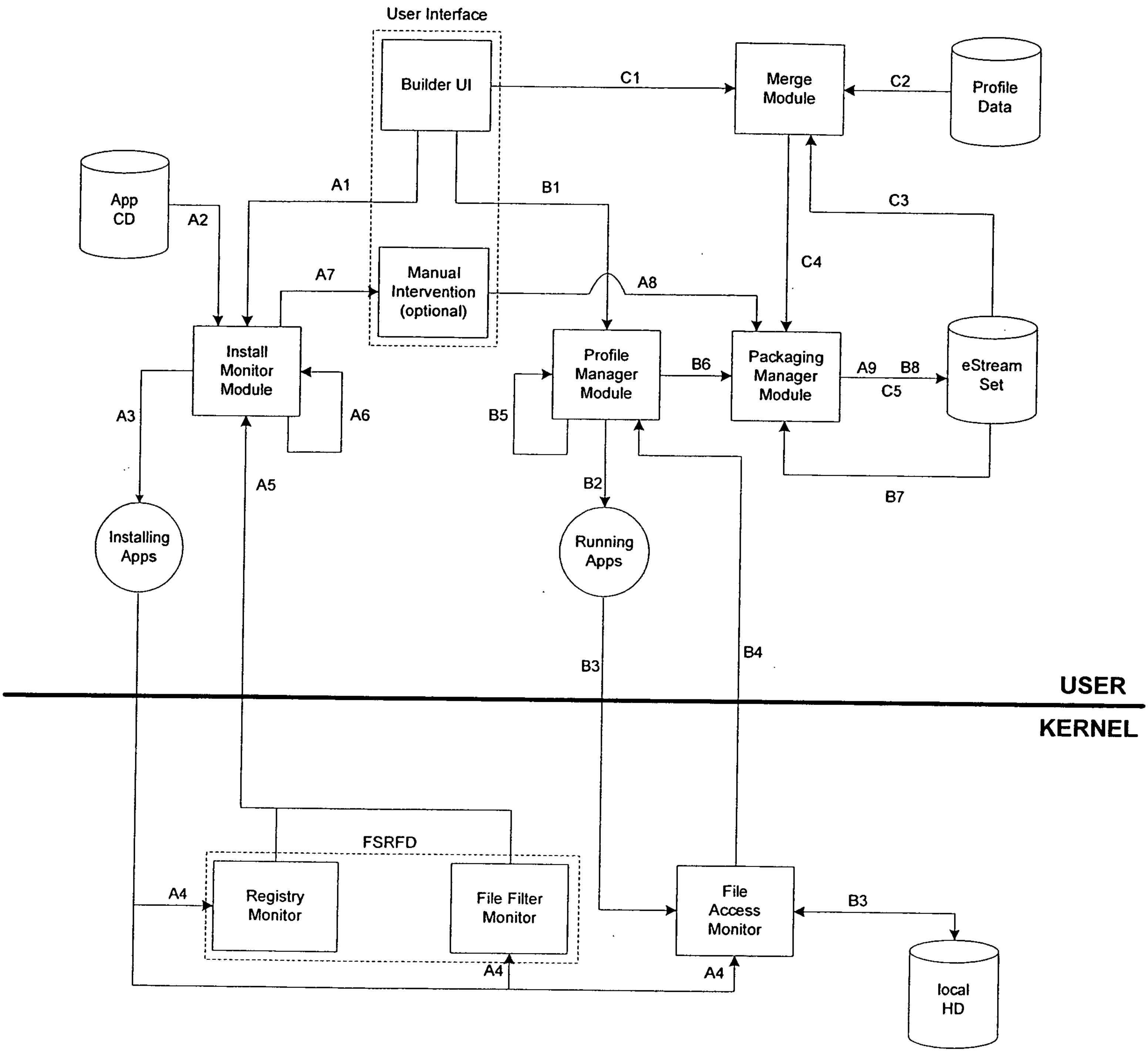
Builder Profiler Control Flow Diagram



Builder eStream Packager Control Flow Diagram



eStream Application Builder High-Level Design Diagram



v 0.1

eStream Builder File Access Monitor Low Level Design

Sanjay Pujare and David Lin
Version 0.1
[redacted]

Functionality

The eStream Application Builder File Access Monitor (FAM) is a kernel-mode device driver that behaves as a file filter driver to has the following responsibilities:

- Monitor any running application's request to access a file or directory
- Track application file and directory accesses
- Track file metadata queries
- Start and stop profiling via IOCTL requests from the user-mode program
- Return the file access data to the user-mode program via I/O Request Packet (IRP)
- Return any error conditions to the user-mode program via IRP

The File Access Monitor is based on the 'Filemon' program. The source code for the program is available free for download over the Web at http://www.sysinternals.com/filemon.htm.

Data type definitions

The File Access Monitor (FAM) monitors a sequence of file block accesses by a particular process or one of its child processes. The FAM also tracks any queries on the file metadata. The combination of the file content and metadata is returned to the Profile Manager for further processing. The following is the data structure externally visible to the other subcomponents outside FAM.

```
Struct SequenceData
{
   UINT NumEntries;
   Struct Entry
   {
      PUNICODE_STRING FilePathName;
      BOOL IsAccessingMetadata;
      ULONG Offset;
      ULONG Size;
   } Entries[NumEntries];
};
```

The FileName contains the null terminating string of the file accessed. IsAccessMetadata flag indicates if the access is on the file metadata or the file content. If the operation is on

the file content, then the fields 'Offset' and 'Size' indicate the location of the read or write operations. Otherwise, the fields 'Offset' and 'Size' are not used.

Interface definitions

Function 1 : Hooks into user defined IOCTL calls

```
// The following is a Fast I/O Device Control
// call interface.  Each of the user IOCTL
// call to the driver is described here.
// The OICTL input and output parameters are
// stored on the IRP on the InputBuffer and
// OutputBuffer respectively.
// This function must be called only by NT I/O
// Manager.

BOOLEAN FileSysFastIoDeviceControl(
   IN PFILE_OBJECT FileObject,
   IN BOOLEAN Wait,
   IN PVOID InputBuffer,
   IN ULONG InputBufferLength,
   OUT PVOID OutputBuffer,
   IN ULONG OutputBufferLength,
   IN ULONG IoControlCode,
   OUT PIO_STATUS_BLOCK IoStatus,
   IN PDEVICE_OBJECT DeviceObject)

Input:
   IoControlCode==IOCTL_FAM_VERSION
      OutputBuffer: version number of the driver

   IoControlCode==IOCTL_FAM_START
      InputBuffer: process ID to monitor

   IoControlCode==IOCTL_FAM_STOP
      OutputBuffer: stop profiling

   IoControlCode==IOCTL_FAM_GETDATA
      OutputBuffer: get sequence data

   IoControlCode==IOCTL_FAM_GETSTATUS
      OutputBuffer: get status from driver


Output:
```

```
Comments:
   The IOCTL calls from the user program to
   the device driver is either through the
   dispatcher or through the Fast I/O
   interface.

Errors:
```

Function 2: DriverEntry

```
// Called by the NT system to initialize
// driver.  The following entries are hooks
// into the OS and are not called by any of our
// component directly.
NTSTATUS DriverEntry(
   IN PDRIVER_OBJECT DriverObject,
   IN PUNICODE_STRING RegistryPath)

Comments:
   Initialize the driver
```

Function 3: Hooks into Fast I/O functions

```
// NT Fast I/O calls.  These are some of the
// hooks into the OS
NTSTATUS FileSysFastIoRead(
   IN PDRIVER_OBJECT DriverObject,
   IN PLARGE_INTEGER FileOffset,
   IN ULONG Length,
   IN BOOLEAN Wait,
   IN ULONG LockKey,
   OUT PVOID Buffer,
   OUT PIO_STATUS_BLOCK IoStatus,
   IN PDEVICE_OBJECT DeviceObject)

Comments:
   Hooks into Fast I/O Read
```

Function 4: Hooks into dispatcher functions

```
// Besides hooks into Fast I/O calls, we
// must also hook into each of the major
// functions like IRP_MJ_CREATE, IRP_MJ_READ,
// etc...
FileSysHookRoutine(
   PDEVICE_OBJECT HookDevice,
```

```
IN IRP Irp)
```

Component design

I/O Hook location

The trickiest part of the File Access Monitor (FAM) component design is determining the locations in the Operating System to hook the routines. FAM must provide driver entry function for initializing the driver. It must also provide hooks into the OS to monitor read and write file operations. In addition, it needs to monitor access to file metadata. And finally, it has to provide the user-mode program a way to communicate to the FAM through the IOCTL calls.

The FAM must behave like any other Windows kernel-mode drivers by exporting the standard *DriverEntry* function. The *DriverEntry* function has the following purposes:

- Check for OS build version.
- Setup the device name
- Call OS routine to create the device
- Make symbolic link to allow device access from Win32 programs
- Create dispatch points for all routines that must be handled
- Setup Fast I/O hooks
- Initialize all data structures

In addition to the *DriverEntry*, the FAM must handle five user defined IOCTL calls.

- IOCTL_FAM_VERSION
- IOCTL_FAM_START
- IOCTL_FAM_STOP
- IOCTL_FAM_GETDATA
- IOCTL_FAM_GETSTATUS

In IOCTL_FAM_START, the handler receives the process ID from the user-mode program. It uses this process ID to filter out relevant file and metadata accesses. In IOCTL_FAM_STOP, the handler stops monitoring and recording any file accesses. In IOCTL_FAM_GETDATA, the handler packages the file access sequence in the I/O Request Packet (IRP) to be returned to the user-mode program. Finally, in IOCTL_FAM_GETSTATUS, the handler returns its current status. This status includes: FAM_STATUS_OK, FAM_STATUS_ERROR, and FAM_STATUS_PROFILING.

In addition to the user defined IOCTL hooks, the FAM must add hook into both the dispatch points and the Fast I/O calls to monitor all read and write requests. In addition, the FAM monitors any metadata accesses. The following is a list of Fast I/O calls it must hook:

- FastIoRead
- FastIoWrite

- FastIoMdlReadComplete
- FastIoMdlWriteComplete
- FastIoReadCompressed
- FastIoWriteCompressed
- FastIoQueryBasicInformation
- FastIoQueryStandardInformation

In the routine to handle FastIoRead and FastIoWrite, the driver must determine the process ID making this request. If the process is in the list of monitoring processes, the file name, file offset, and size is recorded and added to the profile sequence list. In the routine to handle FastIoQueryBasicInformation and FastIoQueryStandardInformation, the driver records the file name associated with this metadata query.

In addition to hooks to the Fast I/O calls, the I/O may call the File System services through standard Windows NT dispatch points. The following is a list of dispatch points to be handled by FAM:

- IRP_MJ_CREATE
- IRP_MJ_READ
- IRP_MJ_WRITE
- IRP_MJ_DIRECTORY_CONTROL + IRP_MN_QUERY_DIRECTORY
- IRP_MJ_QUERY_INFORMATION
- IRP_MJ_SET_INFORMATION
- IRP_MJ_QUERY_EA
- IRP_MJ_SET_EA

The routine to handle IRP_MJ_READ, IRP_MJ_WRITE, and IRP_MN_QUERY_DIRECTORY is handled by the same function as the routine for handling FastIoRead and FastIoWrite. The routine to handle IRP_MJ_QUERY_INFORMATION, IRP_MJ_SET_INFORMATION, IRP_MJ_QUERY_EA, and IRP_MJ_SET_EA are handled by the same function as the routine for handling FastIoQueryInformation.

Communication with user-mode component (Profile Manager)

Besides using the IOCTL to send profile data to the Profile Manager, the FAM must also signal the Profile Manager when new data is available for retrieval. The Profile Manager wakes up from the signal by the FAM and retrieves the information on the blocks of files accessed. FAM also signals the profile manager when the profiled application terminates. FAM uses *KeSetEvent()* to send a 'data available' event signal to the profile manager. Profile manager calls *KeWaitForSingleEvent()* or *KeWaitForMultipleEvent()* to wait for a signal from the kernal-mode driver. *KeClearEvent()* is called by the FAM when the signal to profile manager should be deactivated.

Process Filtering

The FAM must filter the profile information so only relevant data relating to the application under profiled is obtained. This is accomplished by filtering the data according to

the process ID invoking the file access operations. When the FAM is started, the Profile Manager sends its process ID. FAM assumes all child processes of the Profile Manager process ID is to be monitored since the Profile Manager invokes all applications using *CreateProcess()* API. Thus, the new processes all inherit Profile Manager as its parent process ID. The process filtering is accomplished using *PsSetCreateProcessNotify-Routine()* to add a hook to the OS. FAM is notified whenever there is a new process created. The process ID is recorded in a list if its ancestor is the Profile Manager process ID. This list is used to filter the profile data gathered by FAM.

Locks

Since multiple threads may be entering different sections of FAM and accessing different data structures, appropriate locks must be used to prevent multiple threads from reading and writing at the same time. *ExInitalizeResourceLite(), ExAcquireResourceExclusiveLite(),* and *ExReleaseResourceLite()* are used when shared data structure is accessed. These APIs have the requirement that the kernel APCs must be disabled before calling and that IRQL must be lower than DISPATCH_LEVEL. This can be accomplished by using *KeEnterCriticalRegion()* and *KeLeaveCriticalRegion()*. The following is a sample code using these APIs:

```
ERESOURCE gResource; // global variable

KeEnterCriticalRegion()
ExAcquireResourceExclusiveLite(&gResource, TRUE);

<critical section of code>

ExReleaseResourceLite(&gResource);
KeLeaveCriticalRegion();
```

Testing design

- **Unit testing plans**

The plan for unit testing of the FAM consists of using the Profile Manager (PM) and a File Access Driver (FAD) as the test drivers. The PM tests user-defined IOCTL calls. The FAD creates desired data pattern from the OS's I/O Manager to the FAM. The FAD tests the FAM's ability to monitor file accesses by querying files and directories in a particular order. Together, the PM and FAD test coverage of the FAM is complete. The following is a list of tests:

1. Test each user-defined IOCTL interface via PM by sending border cases.
2. Test to make sure FAM captures every file and directory access via standard file I/O requests from a user-mode program called FAD.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

  Cross-component testing for the Builder program is described in the Package Manager low-level design document.

Upgrading/Supportability/Deployment design

Other Builder components log error messages to a predefined file. The kernel-mode programs do not have the capability to read/write to a file. Since FAM is a kernel-mode program, an alternative method of reporting error messages has to be developed. Current, the FAM has a user-defined IOCTL interface (IOCTL_FAM_GETSTATUS) to retrieve the error messages. FAM keeps a stack of error messages encountered and reports the stack of error messages at the request by an appropriate user-mode program.

Open Issues

- Exactly which Fast I/O calls need to be hooked to get all the read and write operations for file accesses?
- Along the same line, which dispatch points need to handled to get all the read and write operations for file accesses?
- Have we hooked into all possible places where the metadata accesses can occur?
- Does the FAM need to hook into FileLock and FileUnlock operations?

eStream Application Builder High-Level Design

Authors: Sanjay Pujare and David Lin

Version 0.1

[redacted]

This document contains the high level design of the eStream Application Builder. The Builder is used to "prepare" an application before it can be eStreamed. This document describes the high level design of the application installation monitoring, file relocation and mapping, gathering of the initial profiling information of an application, the packaging of the eStream Set, and the merging of the newly uploaded user profile data.

Note: all references to "user" should be understood to mean the user of the Builder (i.e. the person who is responsible for creating eStream sets) and not the end-user of eStream technology.

This document described these steps involved in the preparation of the application: Installation Monitoring, Application Profiling, and eStream Packaging.

Modules

Installation Monitor:

- When the application is installed, we need to monitor the installation to see various "things" taking place on the computer. These could be:
  - Various updates to the System Registry
  - Files added to the Install directories (i.e. directories where application bits are copied as specified by the installing user). Lets call this group F_I.
  - Files added/updated to the Shared directories (e.g. "Program Files\Common Files"). Lets call this group F_C.
  - Files added/updated to the System directories (e.g. "WinNT\System32"). Lets call this group F_S.
  - Files added/updated to the User specific directories (e.g. "Documents and Settings\spujare\Application Data"). Lets call this group F_U.

  Note that once this information is gathered by the "Installation Monitor", a single "Installation Set" is prepared where all the files are stored in a single directory hierarchy. Note that files in the F_C, F_S and F_U groups (i.e. F_{CSU} group) are also stored here. For these files a "mapped location" is created under the single directory hierarchy. The Installation Set typically creates a map of all files (called ISM for Installation Set Map) described above with each entry containing the following info:

  1. fileId for the file
  2. location and name of the file. Note that the location will be the actual location for F_I files, but mapped location for the F_{CSU} files.

- After we gather the above information, we need to prepare a "File Relocation Map" (FRM) that is used by the client file spoofer to spoof references to any file in the common file group (i.e. F_{CSU}). For example: when the eStreamed app makes a reference to a file `C:\Program Files\Word\Foobar`, the file spoofer actually redirects that reference to `Z:\Program Files\Word\Foobar`. It does that because of the File Relocation Map. Each entry in the FRM typically has the following info:
  1. fileId (which references an entry in the ISM).
  2. Actual location where the application expects it (i.e. `C:\Program Files\Word\Foobar)`.

Profile Module:
During the application building process, the Builder program queries the user for the name of the application executable. Then Builder program starts and terminates the application executable immediately to gather initial sequence of the application page access pattern. After the initial seed of profile data is acquired, the Profile Sequence Matrix is combined with other appInstallBlock data gathered from the Install Monitor.

Profile Sequence Matrix is a 2D matrix of a profile data. Each entry of the matrix [column C, row R] is an integer value indicating the number of times a page R is requested following the request of page C. This successor request pattern is the page requests missed in the eStream cache manager.

Package Module:
In the final phase of the Builder program, the appInstallBlock is encapsulated into a special installation executable and the application files is archived into a single compressed package. The install executable containing the appInstallBlock and the archive of application files can then be placed in a suitable eStream Set server for ASP to download to their machines.

Merge Module: (not supported in version 1.0)
During normal eStream application usage, the eStream client gathers profile information for that particular run of the application. Then at the termination of an eStream application, it uploads the new Profile Sequence Matrix to the Profile Server. The clients should not upload the Profile Sequence Matrix from previous runs because the Profile Server has no mechanism for distinguishing between previously uploaded data and the newly acquired data.

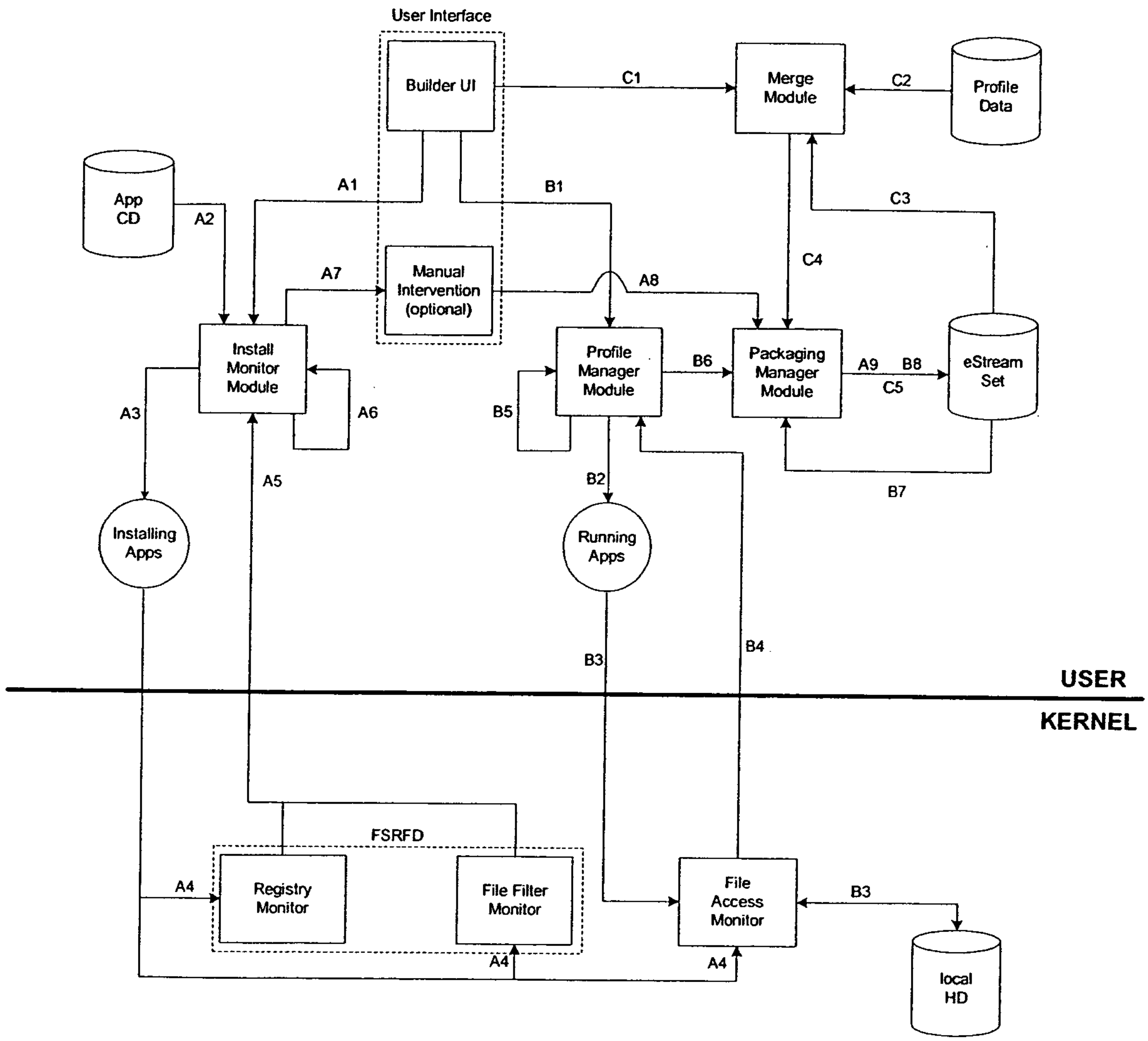
At appropriate time, the Builder is invoked to merge the newly uploaded per-user Profile Sequence Matrix into a collective Matrix. The merging algorithm may be designed with some heuristics to prevent the data biasing toward power users. This collective Matrix can be reinserted into the appropriate appInstallBlock then downloaded by any requesting eStream clients.

Kernel Device Drivers:

In addition, kernel device drivers are used to actually hook into the operating system to monitor the registry and file changes during installation of the application. This is accomplished by the FSRFD module.

The kernel device driver is also used for gathering monitoring file block references from the operating system to the file system. This is accomplished by the File Access Monitor.

eStream Application Builder High-Level Design Diagram



v 0.1

Interfaces

The interfaces are divided into three use cases: application installation monitoring, application profiling, and merging of the uploaded profile data.

Use Case #1: Install Monitor

A1. Builder UI to Install Monitor – send the name of the application installation executable

A2. App CD to Install Monitor – the CD containing the application is fed into the installation monitor module

A3. Install Monitor to Installation App – invoke the installation program

A4. Installation App to FSRFD – monitor all changes to the registry and files when installation program write to local file system

A5. FSRFD to Install Monitor – send all registry and file changes

A6. Install Monitor to itself – repeat all applications in the suite and merge all data

A7. Install Monitor to Manual Intervention – send a list of registry and file captured by the install monitor to UI and allow user to add or delete any entries

A8. Manual Intervention to Package Manager – send the final registry and file relocation data to the packager

A9. Package Manager to database – data set is packaged into appInstallBlock and the rest of the application files suitable for eStreaming

Use Case #2: Profiling

B1. Builder UI to Profile Manager – send the name of the application executable

B2. Profile Manager to Run App – invoke the application

B3. Run App to eStream File Access Monitor – record sequences of page requests

B4. File Access Monitor to Profile Manager – save the profile information

B5. Profile Manager to Profile Manager – repeat for each application in the suite

B6. Profile Manager to Package Manager – send all profile data for merging into a single data

B7. database to Package Manager – get eStream Set from the database

B8. Package Manager to database – save the updated eStream Set

Use Case #3: Merging Profile data (not supported in version 1.0)

C1. Builder UI to Merger – send the application name with profile data to merge

C2. database to Merger – get uploaded Profile Sequence Matrix from the Profile Server

C3. database to Merger – get the old appInstallBlock from database

C4. Merger to Package Manager – reinsert the Profile data into appInstallBlock

C5. Package Manager to database – save the updated appInstallBlock

Requirements

Please see eStream1.0-REQ.doc for the most up-to-date list of the Builder requirements. This requirement list may not contain the most recent changes. Each requirement is identified by a tag such as R-XXXX for easy references elsewhere in the document.

- **R-Background**: The installation monitor runs in the background, when an eStream application is installed as part of its preparation or building.
- **R-RegistryCapture**: The installation monitor captures all the updates to the System Registry that take place during the install. These updates are captured as a .REG file. Note that registry key deletions are also captured and stored in the .REG file. Please see the LLD doc by Charles Booher about the Registry spoofing database.
- **R-FileCapture:** The installation monitor records all the files created in the two kinds of directories: the install directory (the F_I group described above) and the common directories (the F_{CSU} group). All the files created are copied to the Installation Set and the File Relocation Map (FRM) created for the F_{CSU} group files. <Note: as far as a system common DLL is concerned, the eStream client should (a) overwrite the existing DLL if it exists (b) spoof it if doesn't exist. This is necessary because some installations may depend on newer versions of, say MSVCRT.DLL and in Windows there is no way to maintain different versions of the same DLL>.
- **R-InitialProfiling:** The Builder must be able to gather initial set of application profile data. This data consists of the page access pattern for starting and immediately shutting down an application.
- **R-Packaging:** The Builder must package the eStream Set into a easily manageable packages suitable for ASP administrators to download to their servers. The package can be divided into two sets:

  1. Installation Set - an appInstallBlock which is a set of data needed to setup the client machine for running a particular eStream application. The appInstallBlock is converted into an installation executable for simplifying the initial application set-up on the client machine.
  2. Run-time Set - a set of files associated with a particular application. At run-time, appropriate pages from this set of files is streamed to the client.

- **R-Merging (not supported in version 1.0):** The Builder must be able to collect per-user profile data from the Profile Server and merge the profile data into a combined data usable for updating the profile data in the appInstallBlock. This profile data can also be collected for use by the ASP or application developers.
- **R-NoQuietOperation:** The Builder is not required to be run in an environment where no other applications are running. But, since the Builder operates by invoking application installation program, it inherits any restrictive "Quiet-Operation" requirement from the installation program. Thus, if the installation program of an application has a "Quiet-Operation" requirement, then the "Quiet-Operation" must be enforced by the user when running the Builder.
- **R-AllClient**: The Builder should provide functionality to create installation set(s) for each of the clients eStream 1.0 is going to support. <Preferably there should be only one builder program that should recognize the OS it is running on and should create the appropriate installation set. Also if possible, we should be able to "diff" installation sets for different OSs and if they are same, we should be able to create

a single installation set for those OSs. The clients to be supported are W2K, WinNT4.0 and Win98>.

- **R-AppIdGeneration**: It should be possible to change the appId of the eStream set when an ASP wants to "install" the eStream set in order to host it. Typically the builder will generate a default appId number for a new application which can be overridden by the ASP installer by using a Builder tool.
- **R-SuiteSupport**: It should be possible to create a merged eStream set for a suite of applications. E.g. Office consisting of Word, Excel and Powerpoint. This could be done either by providing a tool for merging multiple eStream sets or by allowing the builder to serially monitor multiple installations in a session and then allowing the user to create a single package at the end of the session.
- **R-Testing**: It should be possible to test the Builder using a stand-alone tester and not require the eStream client+server programs.
- **R-UpgradeSupport**: The appInstallBlock should have support for indicating upgrades at the support site. E.g. When an eStream application is upgraded at the server (not as a separate app), the client will no longer be able to access/use it. We should provide some version of the appInstallBlock itself so that clients should detect that they will need to download the appInstallBlock again.
- **R-ManualIntervention**: In the process of creating an eStream set it should be possible for the user to delete file entries and registry entries manually to "trim" the eStream set if she so desires assuming the user knows what she is doing.

Issues

- Profile Sequence Matrix is different for different machine configuration even if the user's usage pattern is the same.
- Profile Sequence Matrix doesn't contain the right successor profile information as eStream cache is warmed up and pages from the cache is replaced.
- Merging Module must take different machine configuration into account. Should this information be uploaded by the client as the same time it uploads the Profile Sequence Matrix to the Profile Server?
- What is the difference between profiling based on the page sequencing seen by the eStream Cache Manager versus the page sequencing missed by the eStream Cache Manager?

eStream Application Builder Interfaces

Authors: Sanjay Pujare and David Lin

Version 0.1

[redacted]

Sub-components of the Builder covered here (note, these are logical divisions and not necessarily physical):

1) Builder UI (the only program directly invoked by the user).
2) Install Monitor module.
3) File System and Registry filter driver (FSRFD).

Interfaces:

1) User to Builder UI:

   There are multiple Use Cases represented by this interface:

   1. Monitor a new install: user needs to provide the (a) setup.exe path and (b) the destination drive where the app will be installed. If this is the second or subsequent install in the session then destination drive should be same as previous ones. After the Builder has finished running setup.exe the user needs to tell the Builder whether the installation was successful or not (unless the Builder can figure it out from the exit code of the setup.exe process). If the installation was not successful, Builder will erase all the installation set data created.

      Validation: Check that the destination drive is not the same as the system drive (e.g. C: on most systems), since the install monitor won't be able to differentiate between common files and installation files in that case.

      Note: The Builder user needs to start with a machine that is "pristine" i.e. this machine should only have the OS installed and no other application or data files. This way we get "maximal" installation to cover all the client configurations i.e. the application install will not be affected by any existing applications or registry settings etc. <At this point it is not clear if "pristine" OS includes any service packs>.

   2. Manual Edits: This allows the user to manually delete file entries and registry entries. Normally this should not be used. This functionality provides a screen that allows entry deletions for registry/files.
   3. Initial Profiling: This allows the initial profile to be created for the application(s) installed in this session. <User needs to provide the .EXE that will be run for the initial profile??>.
   4. Create eStream set: When the user selects this option, the builder creates an eStream set for all the installations done in the session. The Builder

maintains a current appId counter in the registry (not to be confused with the registry monitoring we are doing), and uses that to prompt the user with the appId to be used for this app set. The user can override that appId. The user is also prompted for a version number of the appInstallBlock. The output eStream set is created with the version number. Note: The Builder will NOT compress or encrypt any of the application files. This is done by the eStream server prior to transmitting the files to the client.

2) Builder UI to Install Monitor: Assuming the install monitor is a separate module, there will be an entry function to invoke the install monitor thus:

```
unsigned int InstallMonitor(
        IN  PUNICODE_STRING setup_name,  /* setup.exe path */
        IN PUNICODE_STRING dest_drive,     /* dest drive for install */
        IN PUNICODE_STRING file_for_file_info, /* file for storing file info */
        IN PUNICODE_STRING file_for_rel_info,  /* file for rel info */
        IN PUNICODE_STRING file_for_reg_info, /* file for storing reg info */
        IN BOOL append);               /* append to above 2 files if they exist */
```

<The file formats for the file info, rel file info and the reg info are yet to be fixed>.

3) Install Monitor to setup.exe of the application: The install monitor invokes the setup.exe using CreateProcess or similar Win32 API. Note that it has to invoke this as suspended, since we want to get the process_id and pass it on to the FSRFD so that the FSRFD can start monitoring all the requests for this process as well as its children. After that we want to resume the setup.exe process. (Alternatively, we can just pass the process id of the InstallMonitor process and the setup.exe would be the child of this process).

4) Install Monitor to FSRFD: The interface between these two is going to be very similar to the installmon interface in the prototype. The interface to FSRFD will support the following:

    a) MON_ACTIVATE: Start monitoring. Will pass the process-id, destination drive, system drive

```
DeviceIoControl(hDevice,      // handle to our FSRFD device
                MON_ACTIVATE,  // activate control code
                activateOptions,      // process-id, dest and sys drive
                sizeof(activateOptions),
                0, 0,
                &numBytesReturned,
                0);
```

b) MON_DEACTIVATE: Stop monitoring.

```
DeviceIoControl(hDevice,
        MON_DEACTIVATE,
        0, 0,
        0, 0,
        &numBytesReturned,
        0);
```

c) We would use event objects (using IoCreateNotificationEvent as in the installmon program) for the FSRFD to signal the Install Monitor about the availability of new data. There would be a single event object with the path "\\BaseNamedObjects\installmon" that the FSRFD will use to signal the Install Monitor that a new reg entry or file entry is available.

d) MON_GET_ENTRY: Get the next entry from the FSRFD. This would be either a file or registry update.

```
DeviceIoControl(m_deviceHandle,
                MON_GET_ENTRY,
                0, 0,
                monitorEntry,
                sizeof(monitorEntry),
                &numBytesReturned,
                0);
```

monitorEntry is a struct that is used to convey registry or file updates to the install monitor. The struct tentatively looks as follows:

```
struct MonitorEntry_t {
    UCHAR regOrFile;  // 'R' for registry 'F' for file
    UCHAR addOrDelete; // 'A' for add, 'D' for delete, 'U' for update
    UCHAR valueType; // only registry adds, value type
    ULONG nameLength; // length of name
    ULONG valueLength;  // name of value (when it is a string)
    UNICODE_CHAR  name[1]; // name of nameLength followed by value
                           // valueLength
};
```

Install Monitor Output contents

AppInstallBlock Contents

- **AibVersion**: Magic number or appInstallBlock version number (which identifies the version of the appInstallBlock structure rather than the contents).

- **AppId** (this may be a 64-bit number where the low 32 bits identify app's version number). AppId for Word on Win98 will be different from Word on WinNT (if it turns out that Word binaries are different between NT and 98).
- **VersionNo**: Version number. (this allows us to inform the client that the appInstallBlock has changed for a particular appId). Note: this is different from the low 32 bits of the appId. E.g. Word 97 and Word 98 will be differentiated using the low 32 bits of the appId. Or versions 1.0 and 2.0 of a software might be differentiated using the low 32 bits of the appId. However this field will be used when the Word 97 has been updated using a patch and the old binaries (for the same appId) are no longer available.
- **ClientOSBitMap**: Client OS supported bitmap or ID: for Win2K, Win98, WinNT and other future Oss we might support (it should be possible to say that this appInstallBlock is for more than one OS).
- <**ClientOSServicePack**: We might want to store the service pack level of the OS for which this appInstallBlock has been created. Note that when this field is set we cannot use multiple OS bits in the above field ClientOSBitMap>.
- **ISM**: Installation Set Map (ISM): contains
    - FileId (-1 denotes a deleted file)
    - Full path of the file where the file resides in eStream (e.g. Z:\...)

  Note: a range of fileIds will be reserved for eStream's own files. E.g. the appInstallBlock files could be referenced using these reserved fileIds.
- **FRM**: File Relocation Map (FRM): contains
    - FileId (same as the one in the ISM).
    - The path where the application expects it (e.g. C:\...)

  Note: The client should compare the FRM with the client machine. Any files that exist on the system (such as MSVCRT.DLL) will need to be copied to the client machine (instead of spoofing the file) and the client will need to be told to reboot the system.
- Registry spoof data: contains:
    - HKEY_CURRENT_USER
        - Estream
            - Registry Spoof
                - Add
                    - HKCR
                        - ...
                    - HKLM
                        - ...
                    - HKCC
                        - ...
                - Remove
                    - HKCR
                        - ...
                    - HKLM
                        - ...
                    - HKCC

- Initial profile data: The Builder will profile the application usage to get the sequences of the blocks accessed. The Builder will gather the access sequences sent by the OS to the file system. The profiler will stop when the profile noise exceed some tolerance level (ie. If X percentage of the pages of the application being profiled has been kicked out of memory). A list of entries where each entry has the following format:
  - previous fileID
  - previous blockID (blockID may be replaced with Offset and Length if variable sized cache blocks is supported)
  - next fileID
  - next blockID (blockID may be replaced with Offset and Length if variable sized cache blocks is supported)
  - frequency.
- Initial pages for the app – The size of the initial prefetch blocks will be determined based on the minimum size of the data required to achieve good compression. A list of entries where each entry has the following format:
  - FileID
  - BlockID (blockID may be replaced with Offset and Length if variable sized cache blocks is supported).
- A comment field – this text might be used by the client to show the eStream app user any relevant info
- Special processing needed for this app – this could be a pointer to an EXE/DLL or the code itself in the appInstallBlock or a batch file.

Other Tools Related to the Builder

AppInstallBlock Editor/Modifier

This tool allows you to edit/modify one or all of :

a) the appId
b) the appInstallBlock version number
c) OS bit map (since we may need to add bits for other OSs once we know that binaries are the same for them).

of the appInstallBlock portion of the eStream set created by the builder.

AppInstall Compare Tool

This tool allows one to compare two eStream sets (both the appInstallBlocks and the actual files contained in the file set). Any differences are flagged to the user. This will be useful when we know that an application has the same set of binaries for multiple client OSs.

This tool will also be used to create appUpgradeBlock that will be used by the eStream client to seamlessly upgrade client's apps without necessarily destroying their config files. This is the only way to do it without needing an uninstall of the previous version.

Builder Tester

This program allows us to test the output of the Builder without requiring the eStream client/server set up.

eStream Builder Package Manager Low Level Design

Sanjay Pujare and David Lin
Version 0.1

[redacted]

Functionality

The eStream Application Builder Package Manager is responsible for packaging data gathered from the Installation Monitor, the Profile Manager, and the Upgrade Monitor into a set of data called the eStream Set. For the detail format of the eStream Set, see the separate document on eStream Set.

The Package Manager must perform the following task:

- Create the appInstallBlock containing C-File and Registry data from the Install Monitor; Prefetch data from the Profile Manager; and Updated C-File and Updated Registry data from the Upgrade Monitor
- Create a custom installation DLL needed by a specific applications and add to the appInstallBlock
- Create directory files associated with each directory of the application director and add metadata to the directory
- Create directory files associated with each Windows directory containing both the Spoofed files and Z-files
- Create Concatenated Application File (CAF) which is just a juxtaposition of the application files, eStream directory files, and AppInstallBlock
- Create Size Offset File Table (SOFT) which is a mapping of fileNumber to offset of the start of the CAF file
- Create Root Version Table (RVT) which is a mapping from the version of root to the file number of the root directory file
- Archive the CAF, SOFT, and RVT into a single structure called eStream Set suitable for uploading to the eStream Servers.

Data type definitions

The Package Manager doesn't have any internal data types. It must accept and understand data structures received from the Install Monitor and the Profile Manager. See Install Monitor and Profile Manager components for the description of the data structures.

The Install Monitor is responsible for generating the following list of information: list of copied-files, list of spoof-files, list of files with file numbers, list of add registry entries, and list of delete registry entries. The list of copied-files contains the files copied into

non-application specific directories. The list of spoof-files consists of the files too large to be downloaded to the client in the AppInstallBlock. Those files are copied into some special directory on the Z drive for streaming. The list of files with file numbers consists of the files copied into the standard "Program Files" directory and the files that will be spoofed. The registry information is a list of registry key added or removed during the installation of the application.

```
Struct FileIndexTable {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
      ULONG FileNumber;
   } Entries[NumEntries];
};
Struct FileCopied {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
   } Entries[NumEntries];
}
Struct FileSpoofed {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING OldFilePathName;
      PUNICODE_STRING NewFilePathName;
   } Entries[NumEntries];
};
Struct RegistryInfo {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING KeyName;
      PUNICODE_STRING ValueName;
      PVALUE_DATA ValueData;
   } Entries[NumEntries];
};
Struct IniInfo {
   UINT NumFiles;
   Struct FileEntry {
      PUNICODE_STRING FilePathName;
      UINT NumSections;
      Struct SectionEntry {
         PUNICODE_STRING SectionName;
         UINT NumValues;
         Struct Entry {
            PUNICODE_STRING ValueName;
            PVALUE_DATA ValueData;
         } Entries[NumValues];
      } Entries[NumSections];
```

```
    } Entries[NumFiles];
};
```

The Profile Manager generates AccessCounts and the PrefetchBlocks data with the structures shown below.

```
Struct AccessCounts {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
      ULONG Frequency;
   } Entries[NumEntries];
};
Struct PrefetchBlocks {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
      ULONG BlockNumber;
   } Entries[NumEntries];
};
```

The eStream Set has the following data structure (described in more detail in the separate eStream Set document):

```
Struct eStreamSet {
   Struct eStreamSetHeader header;
   Struct eStreamSetRVT rvt;
   Struct eStreamSetSOFT soft;
   Struct eStreamSetCAF caf;
};
```

Interface definitions

Function 1 : CreateEStreamSet

```
// Create the initial eStream Set from the data
// retrieved from the Install Monitor and the
// Profile Manager.
// This function is called only by the Builder
// UI after data is obtained from Install
// Monitor and Profile Manager.
int CreateEStreamSet(
   IN PFILE_INDEX_TABLE FIT,
   IN PFILE_SPOOFED SpoofFiles,
   IN PFILE_COPIED CopiedFiles,
   IN PREGISTRY_INFO AddRegistry,
   IN PREGISTRY_INFO RemoveRegistry,
   IN PINI_INFO IniInfo,
   IN PACCESS_COUNTS AccessCounts,
   IN PPREFETCH_BLOCKS PrefetchBlocks,
```

```
    IN PVOID DllCode,
    IN PUNICODE_STRING Comment,
    OUT PESTREAM_SET EstreamSet)

 Input:
    FIT: File Index Tree contains the file
      number of the directories, spoofed
      files, and standard files

    CopiedFiles: pointer to a list of files
      To be copied to AppInstallBlock

    SpoofFiles: pointer to a list of files
      To be spoofed on the client

    AddRegistry: pointer to a list of registry
      Data to add
    RemoveRegistry: pointer to a list of
      Registry data to remove
    IniInfo: pointer to a list of ini changes

    AccessCounts: pointer to the list of
      Files with the access frequency

    PrefetchBlocks: pointer to the prefetch data
      To be inserted into the appInstallBlock
      Of the eStream Set

    DllCode: pointer to DLL Code

    Comment: pointer to comment string

 Output:
    EstreamSet: pointer to the eStream Set

 Return Value:
    Success or failure of the packaging process

 Comments:
    The eStream Set will be large for most
    application.  Intermediate data will be
    stored on the local hard-drive.

 Errors:
    OutOfStorage: failure to find enough storage
      For this eStream Set
```

```
FileNotFound: failure to find the files
   Specified by either ListCFiles or
   ListZFiles
```

Function 2 : UpgradeEStreamSet

```
// Upgrade the eStream Set to the latest
// version. This function is only called by
// the Upgrade Manager within the same process.

int UpgradeEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PFILE_INDEX_TABLE UpgFIT,
   IN PFILE_SPOOFED UpgSpoofFiles,
   IN PFILE_COPIED UpgCopiedFiles,
   IN PREGISTRY_INFO UpgAddRegistry,
   IN PREGISTRY_INFO UpgRemoveRegistry,
   IN PACCESS_COUNTS UpgAccessCounts,
   IN PPREFETCH_BLOCKS UpgPrefetchBlocks,
   IN PVOID UpgDllCode,
   IN PUNICODE_STRING UpgComment)

Input:
   UpgFIT: File Index Tree contains the file
      number of the directories, spoofed
      files, and standard files

   UpgCopiedFiles: pointer to a list of files
      To be copied to AppInstallBlock

   UpgSpoofFiles: pointer to a list of files
      To be spoofed on the client

   UpgAddRegistry: pointer to a list of
      Registry data to add

   UpgRemoveRegistry: pointer to a list of
      Registry data to remove

   UpgAccessCounts: pointer to the list of
      Files with the access frequency

   UpgPrefetchBlocks: pointer to the prefetch
      Data to be inserted into the
      AppInstallBlock Of the eStream Set

   UpgDllCode: pointer to DLL Code
```

```
   UpgComment: pointer to comment string

Output:
   EstreamSet: pointer to the eStream Set

Return Value:
   Success or failure of the packaging process

Comments:
   The eStream Set will be large for most
   application.  Intermediate data will be
   stored on the local hard-drive.

Errors:
   OutOfStorage: failure to find enough storage
      For this eStream Set

   FileNotFound: failure to find the files
      Specified by either ListCFiles or
      ListZFiles
```

Function 3 : InsertProfileData

```
// Insert profile and prefetch data into the
// eStream Set. This function is only called by
// the Merge Manager within the same process.

int InsertProfileData(
   INOUT PESTREAM_SET EstreamSet,
   IN PACCESS_COUNTS AccessCounts,
   IN PPREFETCH_BLOCKS PrefetchBlocks)

Input:
   EstreamSet: pointer to old eStream Set
      Before the insertion of the profile
      Data

   AccessCounts: pointer to the list of
      Files with the access frequency

   PrefetchBlocks: pointer to the prefetch data
      To be inserted into the appInstallBlock
      Of the eStream Set

Output:
   EstreamSet: pointer to the new eStream Set

Return Value:
```

```
   Success or failure of the insertion process

Comments:
   The eStream Set will be large for most
   application.  Intermediate data will be
   stored on the local hard-drive.

Errors:
   OutOfStorage: failure to find enough storage
      For this eStream Set

   FileNotFound: failure to find the files
      Associated with the prefetch blocks
```

Component design

The pseudo-code for the function *CreateEStreamSet* is described below:

```
{
    Create AppInstallBlock (AIB) from the following input files:
        o SpoofFiles
        o CopiedFiles
        o AddRegistry
        o RemoveRegistry
        o Prefetch
        o Comment
        o DLLcode
    Assign AppInstallBlock with a unique fileNumber given by the IM;
    Record Root fileNumber in the first entry of Root fileNumber Table (RFT);
    Move AppInstallBlock under the Root directory by adding a new entry in the
        Directory structure;
    Create a Concatenation Application File (CAF) header;
    Create a Size Offset File Table (SOFT) header;
    For each (file in FIT) {
        If (file is a directory) {
            Create the directory with new list of fileNumber, filename, and
                Metadata;
        } Else {
            Find the file in the proper location on the HD;
        }
        Append the file or directory to the end of the CAF file;
```

```
        Append the fileNumber, offset into CAF, and size of file in SOFT;
    }
    Archive CAF, SOFT, and RFT into a single eStream Set;
    Return eStream Set;
}
```

The pseudo-code for the function *UpgradeEStreamSet* is mentioned below:

```
{
    Extract previous version PrevAppInstallBlock from eStream Set;
    Create new AppInstallBlock with new FileNumber;

    Extract PrevSpoofFiles and PrevCopiedFiles from PrevAppInstallBlock;
    Divide the C-Files into SpoofFiles and CopiedFiles;
    Add PrevSpoofFiles to SpoofFiles;
    Add PrevCopiedFiles to CopiedFiles;

    Extract PrevAddRegistry and PrevRemoveRegistry data from
        PrevAppInstallBlock;
    Add any unique ((UpgAddRegistry plus PrevAddRegistry) minus
        UpgRemoveRegistry) in the new AppInstallBlock AddRegistry section;
    Add any unique ((UpgRemoveRegistry plus PrevRemoveRegistry) minus
        UpgAddRegistry in the new AppInstallBlock;

    Add UpgPrefetch data to new AppInstallBlock;
    Add UpgDllCode data to new AppInstallBlock;
    Add UpgComment data to new AppInstallBlock;

    For each (directory in UpgFIT) {
        If (any child fileNumber has changed) {
            Create new directory with updated fileNumber;
            Append file to end of Concatination Application File (CAF);
            Append Size Offset File Table (SOFT) with new entry;
        }
    }
    Append new AppInstallBlock to the end of CAF file;

    Prepend Root FileNumber Table (RFT) with new Root entry;
```

```
        Archive CAF, SOFT, and RFT into a single eStream Set;
        Return eStream Set;
    }
```

The pseudo-code for the function *InsertProfileData* is mentioned below:

```
    {
        // not needed unless merging of uploaded profile data is supported
    }
```

Testing design

This document must have a discussion of how the component is to be tested.

- **Unit testing plans**

  The plan for unit testing Package Manager includes the development of a driver program. This driver interfaces to the Package Manager and invokes the functions with different parameters. The list of possible cases is described below:

  1. Test all interfaces by driving the input parameters with different type of add and remove registry values.
  2. Test all interfaces by driving the input parameters by varying numbers of spoof and copied files.
  3. Test all interfaces by driving the input parameters with some prefetch information.
  4. Test all interfaces for meaningless input values from the IM and PM.
     - Prefetch block containing file number not assigned by IM.
     - IM assigning non-continguous file numbers.
  5. Test upgrade interface for capability to detect and handle bad eStream Set gracefully.
  6. Test upgrade interface and make sure it can detect overlapping file number assignments.
  7. Test upgrade interface and make sure prefetch blocks are not referencing old file number from previous versions.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

  The output data from the Package Manager is called the eStream Set. This eStream Set is the input to a stand-alone test program called the *eStream Extrac-*

tor. The Extractor unpacks and 'install' the eStream Set into the local machine without an eStream client file system installed. This test is used to quickly verify that the eStream Set can be run on a pristine machine. Some of the possible variations of the Extractor test includes:

1. Non-default system variable names. I.e. %SystemRoot% set to "D:\Win" instead of "C:\Winnt".
2. Non-default eStream FS drive letter. Use Y instead of Z.

Upgrading/Supportability/Deployment design

The Package Manager logs all error messages to a predefined file common to all components of the Builder program. Every Builder component prints the error message along with its component name. This allows the user of the Builder program to quickly track down any problem during the Building of a new eStream Set.

Open Issues

- Which Builder component creates the installation DLL when the application needs the custom installation code? Is a new component needed to create the custom DLL separately and insert into AppInstallBlock in the eStream Set as needed?

eStream Builder Profile Manager Low Level Design

Sanjay Pujare and David Lin
Version 0.2
[redacted]

Functionality

The eStream Application Builder Profile Manager is responsible for the following:

- Receive request from the UI Component for one or more application executables to profile.
- Accumulate each run of the profile data in a data structure suitable for merging.
- Invoke each application executable for a fixed amount of time, for a fixed number of prefetch blocks, for a simple start-stop of the program, or for multi-level profiling based on scripts or manual usage of an application.
- Communicate with File Access Monitor (FAM) kernel-mode driver using IOCTLs to start and stop profiling.
- Obtain the complete file access sequence data from the FAM.
- Process the file access sequence into two parts: a table of file access frequency and a list of prefetch blocks.
- Send the resulting profile data to Package Manager component for integration into the AppInstallBlock.

This component will probably exist as a class object and will be instantiated by the Builder user interface component. The component will run in the same process as the user interface component. Please see Builder User Interface component document for more information on that component.

Data type definitions

The Profile Manager imports the file access sequence from the FAM. The data structure is described below: (Please see the File Access Monitor for detail information on the meaning of each field in the data structure)

```
Struct SequenceData
{
  UINT NumEntries;
  Struct Entry
  {
    UNICODE_STRING FilePathName;
    BOOL IsAccessingMetaData;
    ULONG Offset;
    ULONG Size;
  } Entries[NumEntries];
```

```
};
```

Profile Manager creates two data structure from the data received from the Install Monitor and the FAM. The consumer of the output data structure is the Package Manager. These data structures is described in the following two structures:

```
Struct PrefetchBlockList {
   UINT NumSections;
   Struct PrefetchBlocks {
      UINT BlockType;
      UINT NumEntries;
      Struct Entry {
         UNICODE_STRING FilePathName;
         ULONG BlockNumber;
      } Entries[NumEntries];
   } Entries[NumSections];
};
Struct ProfileApplications {
   UINT NumEntries;
   Struct Entry {
      UNICODE_STRING FilePathName;
      UNICODE_STRING Arguments;
   } Entries[NumEntries];
};
```

The access count is used to order the list of files in a directory according to metadata file access frequency. The prefetch data is encorporated into the AppInstallBlock by the Package Manager to be used by the eStream Client.

Interface definitions

Function 1 : StartProfiling

```
int StartProfiling(
   IN PPROFILE_APPLICATIONS AppList,
   IN UINT Type,
   IN UINT TypeData,
   OUT PPREFETCH_BLOCKS PrefetchBlocks)

Input:
   AppList: a list of file pathnames and
      Arguments to run the application.

   Type: type of profiling to do
      SIMPLE: start and stop application
      TIMEBASED: profile for a fixed time and
```

```
        terminate application
      SIZEBASED: profile for a fixed size of
        Profile data

    TypeData: extra data for profile type
      If ProfileType==SIMPLE, TypeData is
        Ignored
      If ProfileType==TIMEBASED, TypeData is
        Length of time in seconds
      If ProfileType==SIZEBASED, TypeData is
        Size of profile data in bytes
      If ProfileType==COMMANDBASED, TypeData is
        Pointer to a possible list of script
        Files to be invoked

    PrefetchBlocks: List of prefetch blocks
      To add to the AppInstallBlock

  Return value:
    Success or failure code of the profiling

  Comments:
    The profile manager component actually send
    IOCTLs to the file filter device driver to
    Start and stop the gathering of the profile
    Data and to retrieve the profile data.

  Errors:
    FileNotFound: some of the application
      Executables in the list may not exist or
      Not readable
    ProfileTimeout: failed to gather the desired
      Size of the profile data after certain
      Amount of time
    DriverFailure: File Access Monitor return
      Failure code to the Profile Manager which
      Must propagate the error to the user
      Interface
```

Component design

Application Invocation

To start profiling, the component must have a list of application pathname to be invoked. The pathname may be an executable with a list of arguments. Or the pathname may be a Windows short-cut file. If the pathname is an exe file, then the standard Win32 API *CreateProcess()* is used. On the other hand, if the file is a Windows short-cut file, then the

component needs to extract relevant information from the short-cut file to make the proper *CreateProcess()* invocation. The following is a pseudo-code for extracting path-name and argument information from the short-cut file using IShellLink interface:

```
GetInfoFromShortCutFile(char *strPath, char *strArg)
{
   IshellLink *psl;
   WIN32_FIND_DATA fd;
   if (SUCCEEDED( CoCreateIstance(CLSID_ShellLink,
                  NULL,
                  CLSCTX_INPROC_SERVER,
                  IID_IShellLink,
                  (LPVOID*) &psl)))
   {
      psl->GetPath(strPath, MAX_LEN, &fd, 0);
      Psl->GetArguments(strArg, MAX_LEN);
      psl->Release();
   }
}
```

Command-based Profiling
One of the options for profiling include the ability to identify blocks of files accessed when specific application command is invoked. The profiler prompts the user for the desired actions (ie. Open document, save document, etc) and gathers file blocks that are accessed corresponding to those actions. These commands are saved into the AppInstall-Block for eStream client to intelligently pick the proper set of blocks to stream. The following is an enumeration of some divisions of prefetch blocks:

- Start Application
- End Application
- Save Document
- Open Document
- Cut Sections
- Copy Sections
- Paste Sections

Communication with kernel-mode driver (FAM)
The Profile Manager communicates with the kernel-mode driver to retrieve the information on the blocks of files accessed. The profile manager waits for a signal from the FAM indicating a new data is available for retrieval. FAM also signals the profile manager when the profiled application terminates. FAM uses *KeSetEvent()* to send a 'data available' event signal to the profile manager. Profile manager calls *KeWaitForSingleEvent()* or *KeWaitForMultipleEvent()* to wait for a signal from the kernal-mode driver. *KeClearEvent()* is called by the FAM when the signal to profile manager should be deactivated.

Pseudo-code

The pseudo-code for the function *StartProfiling* is described below:

```
{
    Initialize GlobalFileAccessCounts and GlobalPrefetchBlocks;
    Load FAM if not loaded;
    For each (executable in the list of application) {
        Start FAM by sending it process ID of the Profile Manager;
        Create new process for this executable and run it;
        Switch (Type of Profiling) {
        Case SIMPLE:
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } Until application start up;
            Break;
        Case TIMEBASED:
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } Until fixed time unit;
            Break;
        Case SIZEBASED:
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } while (size < fixed amount);
            Break;
        Case COMMANDBASED:
            For each command in the list {
                If (script exist)
                    Run script on the executable program;
                Else
                    Prompt operator for proper action;
```

```
                Loop {
                    Wait for an event from FAM;
                    Get Status from FAM;
                    Get SequenceData from FAM;
                } Until script completed;
            }
        }

        Send WM_QUIT message to the application process;
        Loop {
            Wait for an event from FAM;
            Get Status from FAM;
            Get SequenceData from FAM;
        } Until application quit;
        Inform FAM to stop gathering profile data;

        Compute PrefetchBlocks from the SequenceData and append to
            GlobalPrefetchBlocks;
    }
    Unload File Access Monitor (if possible);
    Return GlobalPrefetchBlocks;
}
```

Testing design

- **Unit testing plans**

The plan for unit testing the Profile Manager includes developing a driver to connect to the interface between the Profile Manager and the Builder UI. The driver conducts the following types of tests:

1. Test different type of profiling including simple profiling, time-based profiling, size-based profiling, and script-based profiling.
2. Test different executable programs and make sure the output data "makes sense".
3. Test a list of executables for merging capability of the Profile Manager.
4. Test the interface between Profile Manager (PM) and the File Access Monitor (FAM) using FAM as the test driver. The FAM can check for any valid IOCTL calls from the PM. FAM can also reply to IOCTL calls with different values in the IRP to simulate all possible cases.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

  Cross-component testing for the Builder program is described in the Package Manager low-level design document.

Upgrading/Supportability/Deployment design

The Profile Manager logs all error messages to a predefined file common to all components of the Builder program. Every Builder component prints the error message along with its component name. This allows the user of the Builder program to quickly track down any problem during the Building of a new eStream Set.

Open Issues

- How to automate profiling so the application doesn't require any intervention? Look into using Rational TestSuite programs.
- Can a subset of Winstone be used for profiling? How do we determine which part of the profile data is more useful to the end-user?
- Should Profile Manager actually create data structures like PrefetchBlocks which require FileNumber assignments? Or should Profile Manager just create the output data without knowning FileNumbers? Then Package Manager associates the file numbers assigned by the Install Monitor with the profile data gathered by the Profiler.

Tricky Builder Issues

Author: Sanjay Pujare

This document enumerates all those tricky issues that may make the Builder's job difficult. Even though some solution may be proposed for some issues, not every issue would have a solution described in this document. The purpose of this document is mainly to keep track of Builder issues that may impose some limitations on the eStream technology. This way Omnishift marketing and deployment are aware of these limitations.

1) The Builder cannot capture updates to existing files in an intelligent fashion (i.e. if the updates are based on a context or existing contents, it is very difficult to capture that). So the current Builder will just flag an error, if such an update occurs.

   Solution
   - These updates are probably very rare, so we can defer it to the next release.
   - For this release, we can try to solve this on a case by case basis e.g. we will try to solve this issue for INI files.
   - Based on our understanding of general app installations, we might be able to make some generalizations that we can use in eStream e.g. Only certain files get updated; there is a definite pattern of updates.
2) The current Builder drivers are based on the NT driver model, and hopefully we can implement the same functionality in the Win98 drivers, but this needs to be ensured. (This shouldn't be an issue, but…)
3) We need to think some more about those cases when device drivers are installed by apps. Issues that can arise:
   - This may not work correctly on the eStream client, just because the driver installation didn't take place properly.
   - The Builder would need to be able to figure out in an automated way if a client reboot is required or not.
   - If the driver installation is h/w or s/w specific that can be difficult to tackle.

   Solution
   - As we eStream more apps and gain more experience, we should be able to figure out solutions.
4) There could be an ambiguity when the Installmon is trying to change absolute paths (or absolute values in general) to relative paths. e.g. A path like C:\WINNT can be changed to %SystemRoot% or %windir% since both of those environment variables are set to "C:\WINNT" on my system.

   Solution
   - We can prioritize env vars and registry keys as described in the BuilderUI-LLD design document.

- We should encourage Builder operators to use as distinct values as possible for env variables and registry keys for the Builder machine.

eStream BuilderUI Low Level Design

Sanjay M Pujare
<Date>

Functionality

The BuilderUI is the user interface part of the Builder. The operator uses this interface to use various functions provided by the Builder. Note that this UI may or may not be a graphical user interface. This low-level design is based on the assumption that a graphical user interface is not necessary.

Data type definitions

Interface definitions

Component Design

When the Builder UI is invoked with command line arguments which indicate that this was invoked by the `Runonce` mechanism of Windows, the control is transferred to the `InstallMon::startCaptureAfterReboot()` function with the command line arguments passed as arguments to the function. When the Builder is invoked normally, it presents a menu which is managed by the function `MainMenu`.

MainMenu

```
This function manages the following menu hierarchy. Each
menu option (leaf node) is followed by a function name in
parentheses that is called to handle the option.
1) eStream Set Menu
   1) New eStream Set (NewEStreamSet)
   2) Open eStream Set (OpenEStreamSet)
   3) Save New/Upgraded eStream Set (EstreamSetCreation)
2) Monitoring Menu
   1) Start Monitor (StartMonitor)
   2) Stop Monitor (StopMonitor)
   3) Check Status (CheckStatus)
   4) Inform Machine Reboot (InformMachineReboot)
   5) Get and Resolve Registry Set (GetRegistrySet)
   6) Get and Resolve Files Set (GetFilesSet)
3) Profiling Menu
   1) Set the location of app executable (GetAppPath)
   2) Gather Initial Profile (GatherInitialProfile)
4) eStream Set Creation Menu
   1) Set custom DLL (GetCustomDLL)
   2) Set User Comment (GetUserComment)
   3) Set environment variables (GetEnvVars)
```

```
4) Set Reboot flag (GetRebootFlag).
5) Set License Agreement (GetLicenseAgreement).
```

NewEStreamSet

```
{
  If there is an existing eStream set that hasn't been
  saved, warn the user.
  Get the following values from the user:
    • Name of the app setup program in gAppSetup
    • Dest directory where app will be installed in gDest-
      Dir (provide a default value).
    • Dest location to store the new eStream set in
      gDestEstreamPath (provide a default value).
}
```

OpenEStreamSet

```
{
  If there is an existing eStream set that hasn't been
  saved, warn the user.
  Get the following values from the user:
  • Location of the existing eStream set in gSrcEstream-
    Path (provide a default value).
  • Whether the user wants to create an upgrade from this,
    or just wants to change the existing eStream set (gUp-
    grade)
  • If this is an upgrade (gUpgrade is true), get all the
    values obtained by NewEstreamSet (i.e. gAppSetup,
    gDestDir, gDestEstreamPath).

  Read the eStream set pointed to by gSrcEstreamPath;
  Load the existing file tables in gSrcCopiedFiles,
  gSrcSpoofedFiles and gSrcEFSFiles arrays;
}
```

EstreamSetCreation

```
{
  If there is no working eStream set, give error and re-
  turn.
  if (gUpgrade) {
    Call UpgradeEStreamSet() with appropriate arguments;
  }
  else {
    Call CreateEStreamSet() with appropriate arguments;
  }
  if (gDestEstreamPath is not set) {
    assert(this is an update of an existing eStream set
```

```
      and not an upgrade or a new eStream set creation);
    gDestEstreamPath = gSrcEstreamPath;
  }
  Save the eStream Set from memory to file to
    gDestEstreamPath;
}
```

StartMonitor

```
{
  If there is no working eStream set, give error and re-
  turn.
  if (gUpgrade) {
    Combine the gSrcSpoofedFiles and gSrcEFSFiles into an
    array fileTableArray as expected by startCapture
    below;
  }
  Call InstallMon::startCapture(gAppSetup, gDestDir,
       gUpgrade, fileTableArray);
}
```

StopMonitor

```
{
  If monitoring wasn't started, give error and return.
  Call InstallMon::stopCapture();
}
```

CheckStatus

```
{
  Ensure that monitoring was started and not stopped
  Call InstallMon::checkSetupStatus();
}
```

InformMachineReboot

```
{
  Ensure that we are in the middle of monitoring.
  Call InstallMon::machineToBeRebooted();
}
```

GetRegistrySet

```
{
  Call InstallMon::getRegistryList();
  Store the set in gNewRegistry data structure;
}
```

GetFilesSet

```
{
  Call InstallMon::getFilesList();
```

```
    Store the set in gNewFiles data structure;
    Also we need to capture changes made to INI files; since
    this has not been taken care of in the app install block
    and other parts of the Builder+Client we will need to
    make changes in all those components which are affected.
}
```

GetAppPath

```
{
    Ensure that all the InstallMon related data was captured.
    Get the location of the app that needs to be run
    to gather profiling info;
    Store it in gProfileAppPath;
}
```

GatherInitialProfile

```
{
    // this is the function that is used to get the initial
    // profile data (i.e. set of pages prefetched when an
    // eStream app is started for the first time)
    // implementation of this is yet to be defined
}
```

GetCustomDLL

```
{
    Get the location of the custom DLL file, validate that it
    is a DLL and store the path in gCustomDLLPath;
}
```

GetUserComment

```
{
    Get the user comment (optionally by browsing a text file)
    and store it in gUserComment;
}
```

GetEnvVars

```
{
    Call the InstallMon::setEnvVars() function;
}
```

GetRebootFlag

```
{
    Until we come up with an algorithm to determine if a re-
    boot is required for an eStream app, get this value from
    the user. The default is FALSE: we do not want to reboot
    the client PC when the user subscribes to this eStream
    app.
```

}

GetLicenseAgreement

```
{
  Get license agreement from the user. This could be:
    □ either, a default OmniShift license agreement
    □ or, a default ASP agreement
    □ or, license agreement that the app displayed.
  Let the user decide and enter the proper one. Provide a
  default based on our policy.
}
```

Interesting issues to deal with:

Testing design

Unit testing plans

Testing of the UI itself is a comparatively trivial task. The testing will basically consist of traversing the whole menu hierarchy. Since the menu is similar to a typical File Open -> File Edit -> File Save kind of a user application, this can be tested using simple hooks.

Stress testing plans

Since the Builder will be used in house at least initially only simple stress testing should be necessary. Make sure that Builder doesn't crash in the middle of processing so that we don't lose important data. Performance is not considered to be important.

Coverage testing plans

Basically the following 3 paths will be exercised:

1. Create a new eStream set
2. Open an existing eStream set to modify some data in it
3. Open an existing eStream set to create an upgrade for it

Cross-component testing plans

Will be tested as a component of the whole builder.

Upgrading/Supportability/Deployment design

Deployment: This will be used in-house, so no deployment considerations.

Open Issues

- We need to think about the problem of converting absolute file paths discovered in the monitoring process to paths relative to some application or system registry key. Although most cases may not present a problem, we may have some difficult cases, which may make this problem non-automatable i.e. we may need some user intervention. Consider the case:

  KEYONE = C:\FOO\BAR

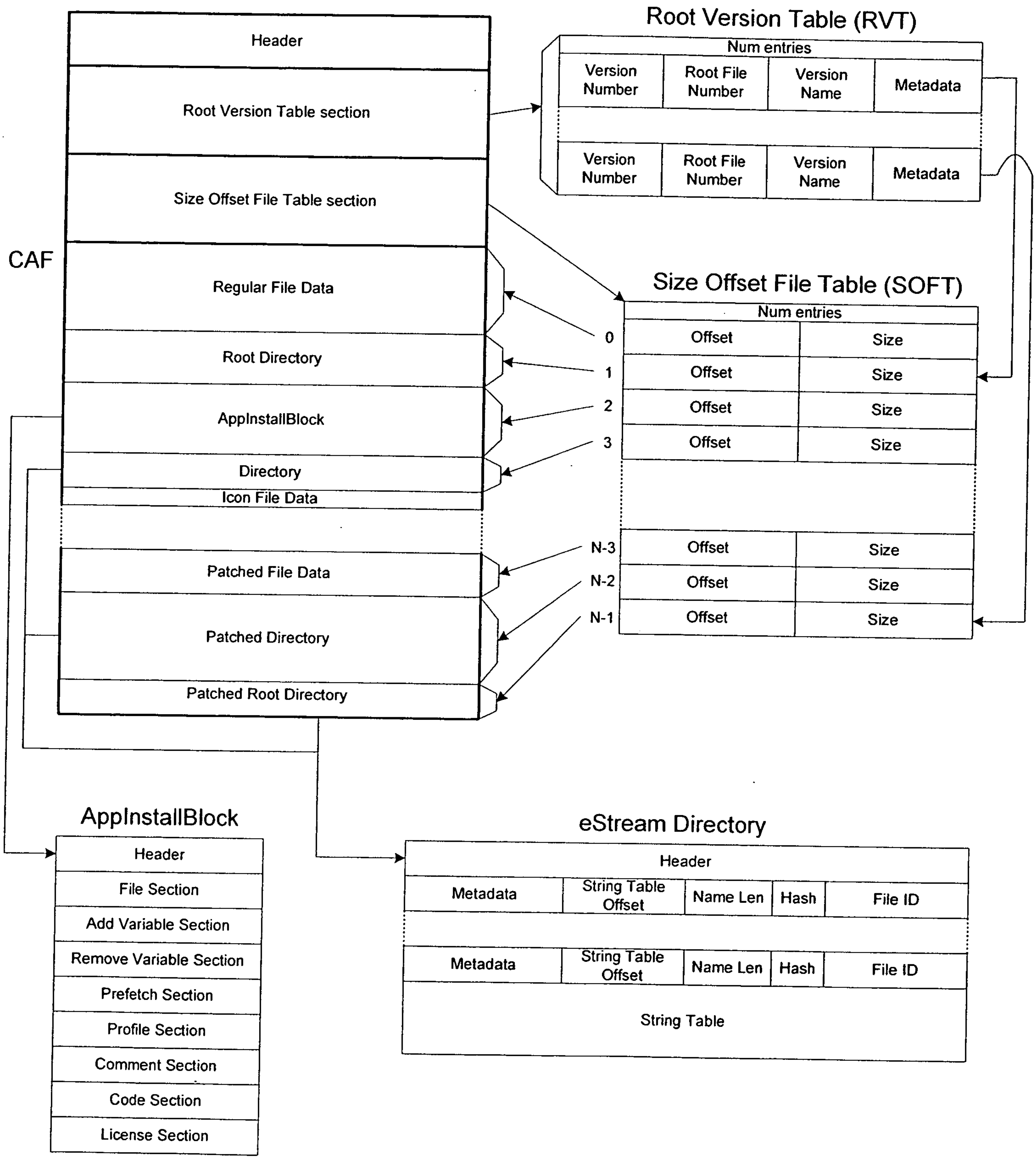
  KEYTWO = C:\FOO

  KEYTHREE = BAR

  If we notice that a file was copied to C:\FOO\BAR it won't be possible to convert this absolute path to a unique relative path since there are 2 solutions possible: %KEYONE% or %KEYTWO%\%KEYTHREE%.

  The way to solve this is by tracking only a set of well-known environment variables and registry keys. Also in this set we prioritize all of them. So in the above case, %KEYONE% will be preferred over %KEYONE%\%KEYTHREE% just because of the way they were prioritized.

Format of the eStream Set



v 0.2

eStream Set Format Low Level Design

Sanjay Pujare and David Lin
Version 0.3
[redacted]

Functionality

The eStream Set is a data set associated with an application suitable for streaming over the network. The eStream Set is generated by the eStream Builder program. This program converts locally installable applications into the eStream Set. This document describes the format of the eStream Set.

Note: Fields greater than a single byte is stored in little-endian format. All strings are in Unicode unless specifically stated otherwise. The eStream Set file size is limited to 2^64 bytes.

Data type definitions

The format of the eStream Set consists of 4 sections: header, Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF) sections.

1. Header section

- **MagicNumber [4 bytes]:** Magic number identifying the file content with the eStream Set
- **ESSVersion [4 bytes]:** Version number of the eStream Set format.
- **AppID [16 bytes]:** A unique application ID for this application. This field must match the AppID located in the AppInstallBlock. Guidgen is used to create this identifier.

- **RVToffset [8 bytes]:** Byte offset into the start of the RVT section.
- **RVTsize [8 bytes]:** Byte size of the RVT section.
- **SOFToffset [8 bytes]:** Byte offset into the start of the SOFT section.
- **SOFTsize [8 bytes]:** Byte size of the SOFT section.
- **CAFoffset [8 bytes]:** Byte offset into the start of the CAF section.
- **CAFsize [8 bytes]:** Byte size of the CAF section.

- **VendorNameLength [2 bytes]:** Byte length of the vendor name.
- **VendorName [X bytes]:** Name of the software vendor who created this application. I.e. "Microsoft". Null-terminated.
- **AppBaseNameLength [2 bytes]:** Byte length of the application base name.
- **AppBaseName [X bytes]:** Base name of the application. I.e. "Word 2000". Null-terminated.

- **MessageLength [2 bytes]:** Byte length of the message text.
- **Message [X bytes]:** Message text. Null-terminated.

2. Root Version Table (RVT) section

The Root version entries are ordered in a decreasing value according to their file numbers. The Builder generates unique file numbers within each eStream Set in a monotonically increasing value. So larger root file number implies later versions of the same application. The latest root version is located at the top of the section to allow the eStream Server easy access to the data associated with the latest root version.

- **NumberEntries [4 bytes]:** Number of patch versions contained in this eStream Set. The number indicates the number of entries in the Root Version Table (RVT).

Root Version structure: (variable number of entries)

- **VersionNumber [4 bytes]:** Version number of the root directory.
- **FileNumber [4 bytes]:** File number of the root directory.
- **VersionName [32 bytes]:** Application version name. I.e. "SP 1".
- **Metadata [32 bytes]:** See eStream FS Directory for format of the metadata.

3. Size Offset File Table (SOFT) section

The SOFT table contains information to locate specific files in the CAF section. The entries are ordered according to the file number starting from 0 to NumberFiles-1.

- **NumberFiles [4 bytes]:** Number of entries in this section.

SOFT entry structure: (variable number of entries)

- **Offset [8 bytes]:** Byte offset into CAF of the start of this file.
- **Size [8 bytes]:** Byte size of this file. The file is located from address Offset to Offset+Size.

4. Concatenation Application File (CAF) section

CAF is a concatenation of all file or directory data into a single data structure. Each piece of data can be a regular file, an AppInstallBlock, an eStream FS directory file, or an icon file.

a. *Regular Files*

- **FileData [X bytes]:** Content of a regular file

b. *AppInstallBlock (See AppInstallBlock document for detail format)*

A simplified description of the AppInstallBlock is listed here. For exact detail of the individual fields in the AppInstallBlock, please see AppInstallBlock Low-Level Design document.

- **Header section [X bytes]:** Header for AppInstallBlock containing information to identify this AppInstallBlock.
- **Files section [X bytes]:** Section containing file to be copied or spoofed.
- **AddVariable section [X bytes]:** Section containing system variables to be added.
- **RemoveVariable section [X bytes]:** Section containing system variables to be removed.
- **Prefetch section [X bytes]:** Section containing pointers to files to be pre-fetched to the client.
- **Profile section [X bytes]:** Section containing profile data. (not used in eStream 1.0)
- **Comment section [X bytes]:** Section containing comments about AppInstallBlock.
- **Code section [X bytes]:** Section containing application-specific code needed to prepare local machine for streaming this application
- **LicenseAgreement section [X bytes]:** Section containing licensing agreement message.

c. *EStream Directory*

An eStream Directory contains information about the subdirectories and files located within this directory. The information includes file number, names, and metadata associated with the files.

- **MagicNumber [4 bytes]:** Magic number for eStream directory file.
- **StringTable [4 bytes]:** Byte size offset to beginning of the string table.
- **StringTableLength [4 bytes]:** Byte size length of the string table.
- **ParentFileID [16+4 bytes]:** AppID+FileNumber of the parent directory. AppID is set to 0 if the directory is the root.
- **SelfFileID [16+4 bytes]:** AppID+FileNumber of this directory.
- **NumFiles [4 bytes]:** Number of files in the directory.

Fixed length entry for each file in the directory:

- **FileID [16+4 bytes]:** AppID+FileNumber of each file in this directory.
- **NameHash [4 bytes]:** Hash value of the file name. Algorithm TBD.

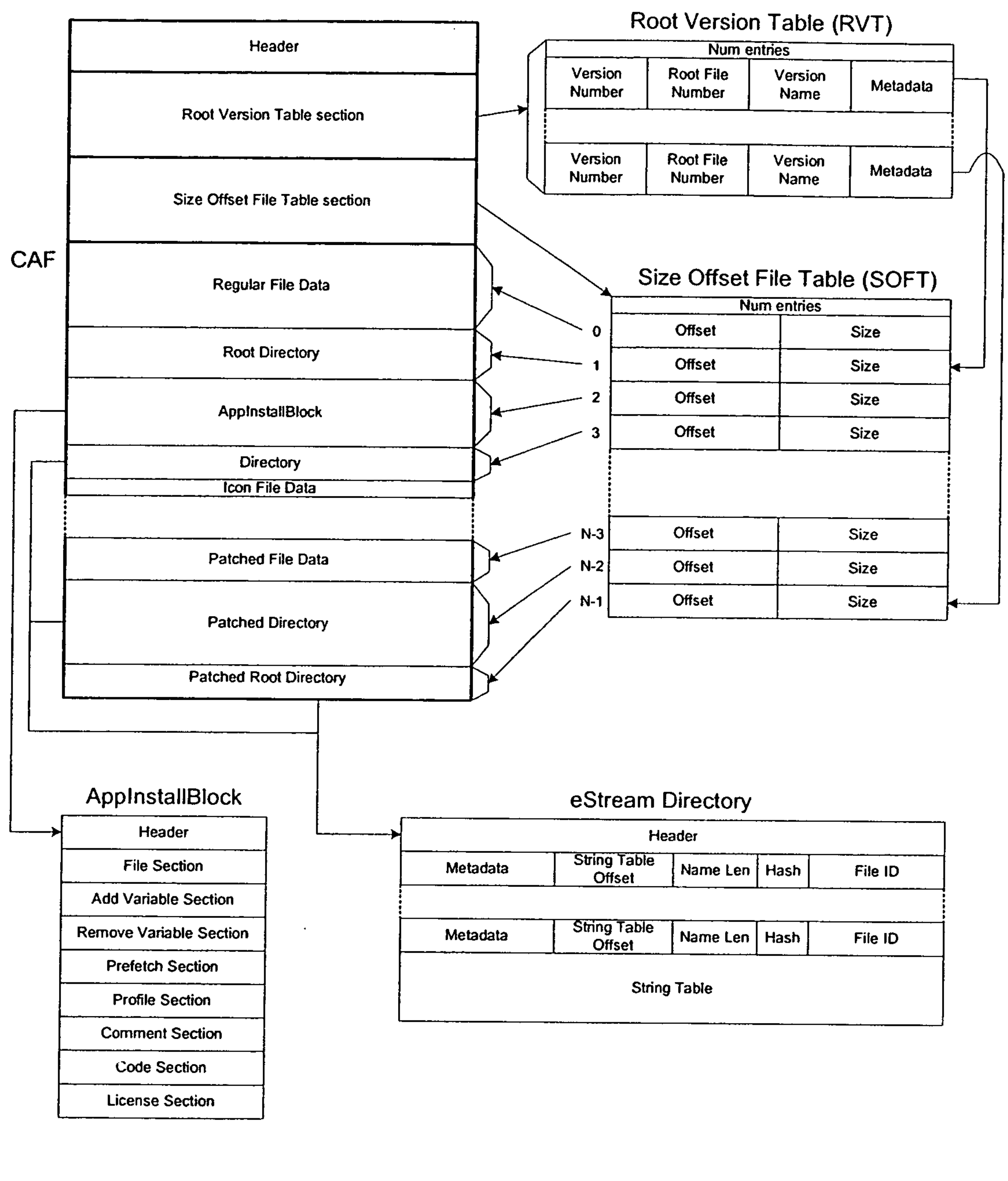
- **FileNameOffset [4 bytes]:** Offset where the file name is located, relative to the beginning of the string table.
- **FileNameLength [4 bytes]:** Byte size length of the file name that is null-terminated.
- **Metadata [32 bytes]:** The metadata consists of file **byte size** (8 bytes), file **creation time** (8 bytes), file **modified time** (8 bytes), **attribute flags** (4 bytes), **eStream flags** (4 bytes). The bits of the **attribute flags** have the following meaning:
  - **Bit 0:** Read-only – Set if file is read-only
  - **Bit 1:** Hidden – Set if file is hidden from user

  The bits of the **eStream flags** have the following meaning:
  - **Bit 0:** ForceUpgrade – Used only on root file. Set if client is forced to upgrade to this particular version if the current root version on the client is older.
  - **Bit 1:** RequireAccessToken – Set if file require access token before client can read it.
  - **Bit 2:** IsDirectory – Set if the file is a eStream Directory.

d. Icon files

- **IconFileData [X bytes]:** Content of an icon file.

Format of the eStream Set



Open Issues

- Where is the metadata associated with the Root directory located? Currently, root metadata is located in the root version table. All other files and directory metadata can be found in their parent directory.

eStream FSRFD Low Level Design

Sanjay M Pujare
[redacted]

Functionality

The File System and Registry Filter Driver (FSRFD) is a part of the Builder module that monitors file system and registry updates initiated by the Builder process. This driver just intercepts such requests and records them and returns the recorded data to the Install Monitor (*INSTALLMON* described in another LLD) program when requested by the latter. All the intelligence, such as any decision-making logic, resides in the INSTALLMON.

For registry updates such as add or modify, the FSRFD needs to record the value added or modified. For registry deletes only the value name needs to be recorded. For file updates, there is no need to record the file contents added or modified, since the Builder would be interested only in the final contents of a file.

Note:

1. **This does not cover those rare cases where an existing file is updated by an application install, and the eStream client would need to make the same updates. This kind of functionality is difficult to implement and will not be considered for 1.0.**
2. **The FSRFD will be used to monitor only one install at a time to simplify the design of the FSRFD. That means you cannot invoke multiple instances of the Builder at a time to monitor multiple installations. All Builder invocations on a machine have to be strictly sequential.**
3. **This design is based on the driver model for WinNT and Win2K. The Win98 driver is not covered here (yet).**

Data type definitions

The following struct is used to communicate information related to activating the FSRFD. Specifically, the process-id of the INSTALLMON and the 2 drives whose accesses need to be monitored are passed.

```
struct MonitorActivate_t {
    ULONG processId; // PID of INSTALLMON
    UCHAR sysDrive;  // System drive letter
    UCHAR destDrive; // Dest drive letter
};
```

The following struct is used to return monitored data back to the INSTALLMON. Note that this is a variable size struct where the last field keyName is an array of one wide-char, but in reality is an array of length whose value is the sum of 3 length fields in the struct (nameLength, valueNameLength, and dataLength).

```
struct IMON_ENTRY {
    UCHAR regOrFile; // 'R' for registry, 'F' for files
                     // and 'E' for end of data
    UCHAR updateType; // 'A' for add, 'D' for delete,
                      // 'U' for update
    UCHAR valueType;  // for registry only: value type
// REG_SZ, REG_DWORD, REG_BINARY,
// REG_DWORD_LITTLE_ENDIAN, REG_DWORD_BIG_ENDIAN,
// REG_EXPAND_SZ, REG_LINK, REG_MULTI_SZ, REG_NONE,
// REG_QWORD, REG_QWORD_LITTLE_ENDIAN,
// REG_RESOURCE_LIST
    ULONG nameLength; // length of name (file or
                      // registry key) in wchars
    ULONG valueNameLength; // length of value name (if
                           // it exists) in wchars
    ULONG dataLength;     // length of data in bytes
    WCHAR keyName[1];    // keyName followed by
                        // valueName followed by
                        // data: note none of these are
                        // null terminated & are wide
                        // chars
};
```

Note about the updateType field: 'A' is used for file creation and 'U' is used for any updates to the file. So if a 'U' is seen without an 'A' for a file that means the file was modified but not created in this session.

The following struct is used as the device extension in the FSRFD devices. Note that this extension is used for all device objects: the device that is created in the DriverEntry function to represent an "INSTALLMON" device for the INSTALLMON to access our driver as well as the devices that are created to create a filter layer above existing drives.

```
enum DEVICE_TYPE {
        INSTALLMONINTERFACE,
        STANDARD
};
```

```
struct DeviceExtension_t {
            PDEVICE_OBJECT deviceObject; // device
                           // for lower layer
            DEVICE_TYPE type; // see design
            KMUTEX pDeviceMutex;
            ProcessIdList
            // pointer to list of process-ids we are
            // interested in monitoring
            EntryList
            // This is the list that stores all the
            // info captured by the FSRFD until each
            // entry is queried by INSTALLMON
};
```

There is an array for FastIo that stores all the FastIo function pointers which is required in a file system filter driver such as this. Note that we need to provide entry points for all (or most?) of the FastIo routines in the dispatch table, since we need to pass the request down to the lower layer driver even if we are not interested in intercepting the request. For only some of the requests (e.g. all the FASTIO_*WRITE* requests), we would be recording the file access and creating an entry to be returned to INSTALLMON. The Component Design section describes in more detail the FastIO routines that are implemented by this driver.

Interface definitions

INSTALLMON interfaces

Since the FSRFD is a driver, it cannot provide directly callable APIs. Instead the INSTALLMON communicates with the FSRFD using the DeviceIoControl Win32 API. It uses Ioctl codes MON_ACTIVATE, MON_DEACTIVATE and MON_GET_ENTRY. The DeviceIoControl API looks as follows:

```
BOOL DeviceIoControl(
  HANDLE hDevice,              // handle to device
  DWORD dwIoControlCode,       // operation control code
  LPVOID lpInBuffer,           // input data buffer
  DWORD nInBufferSize,         // size of input data buffer
  LPVOID lpOutBuffer,          // output data buffer
  DWORD nOutBufferSize,        // size of output data buffer
  LPDWORD lpBytesReturned,     // byte count
  LPOVERLAPPED lpOverlapped    // overlapped information
);
```

The semantics of each of the MON_* codes is defined below. Note that this is described from the caller's (i.e. INSTALLMON) point of view.

Ioctl 1 – MON_ACTIVATE

Input:

hDevice:
is the handle to the FSRFD device created.

dwIoControlCode:
The value MON_ACTIVATE

lpInBuffer:
Address of MonitorActivate_t struct. This contains the process id of INSTALLMON.

nInBufferSize:
Sizeof(MonitorActivate_t)

Output:

lpOutBuffer:
Should be a ptr to a ULONG (at least).

nOutBufferSize:
Size of above buffer

lpBytesReturned:
Should be a ptr to a DWORD where byte count of data returned in lpOutBuffer is returned.

Comments:

MON_ACTIVATE is sent to the FSRFD when the INSTALLMON wants to start monitoring an installation. MON_ACTIVATE can be sent only when the FSRFD is not already active - either after the driver is loaded the first time or after the last MON_DEACTIVATE request.

Errors:

- STATUS_ALREADY_ACTIVE: FSRFD was already activated. The processId of the old activation is returned in the ULONG pointed by lpOutBuffer.
- STATUS_INVALID_ARG: One of the arguments passed is not valid (either invalid MonitorActivate_t ptr or processId).

- STATUS_INVALID_DRIVE: Either the sysDrive or the destDrive (or both) is invalid.

Ioctl 2 – MON_DEACTIVATE

Input:

hDevice:
is the handle to the FSRFD device created.

dwIoControlCode:
The value MON_DEACTIVATE

lpInBuffer:
NULL, or pointer to a ULONG where the ULONG has a non-zero value indicating a "forced" deactivation. A "forced" deactivation is done when the FSRFD still has entries that are not going to be retrieved.

nInBufferSize:
0 or size of ULONG (depending on the above).

Output:

No need for OUT arguments.

Comments:

MON_DEACTIVATE is sent to the FSRFD when the INSTALLMON wants to stop monitoring an installation. MON_DEACTIVATE can be sent only when the FSRFD is already active i.e. after the last MON_ACTIVATE request.

Errors:

- STATUS_NOT_ACTIVE: FSRFD was already deactivated. No special action is needed to be taken for this error condition. The caller can simply send a new MON_ACTIVATE.
- STATUS_PENDING_DATA: The FSRFD has some entries that were not read using

MON_GET_ENTRY. This error is only returned in case of non-forced deactivation.

Ioctl 3 – MON_GET_ENTRY

Input:

hDevice:
is the handle to the FSRFD device created.

dwIoControlCode:
The value MON_GET_ENTRY

lpInBuffer:
NULL.

nInBufferSize:
0

Output:

lpOutBuffer:
Should be a ptr to a IMON_ENTRY struct.

nOutBufferSize:
Size of above buffer

lpBytesReturned:
Should be a ptr to a DWORD where byte count of data returned in lpOutBuffer is returned.

Comments:

MON_GET_ENTRY is sent to get the next "entry" from the FSRFD. An "entry" is a record of a registry or file update intercepted by the FSRFD. The details are returned in the IMON_ENTRY struct passed in the lpOutBuffer argument. Note that the FSRFD uses an event object (created using the IoCreateNotificationEvent API) to signal the INSTALLMON that an "entry" is available to be read. INSTALLMON waits on this event object before retrieving the entry using MON_GET_ENTRY.

Errors:

- STATUS_NOT_ACTIVE: FSRFD was not activated.
- STATUS_INVALID_ARG: One of the pointer arguments passed is not valid.
- STATUS_INSUFF_BUFFER: The buffer size indicated by nOutBufferSize is insufficient for the current "entry" data.

Ioctl4 – MON_GET_ERROR

We need this to indicate occurrence of an error whenever this occurs in any of the dispatch functions. We can either implement this control code or just return an error code for any MON_GET_ENTRY call that reflects that an error occurred.

Event Object interface

As mentioned above, an event object (lets call it IMON event object) will be used to signal the INSTALLMON that a new entry is available. This event object will be created in the FSRFD in the processing of MON_ACTIVATE ioctl, as:

```
ext->pEvent = IoCreateNotificationEvent(
    L"\\BaseNamedObjects\\INSTALLMONEVENT",
          ext->eventHandle);
```

Whenever the FSRFD has a new entry, it signals using the above event object as follows:

```
KeSetEvent(ext->pEvent, 0, FALSE);
```

Whenever INSTALLMON gets the next entry using MON_GET_ENTRY ioctl, the FSRFD resets the event (if the list of entries is going to be empty after this entry) as follows:

```
KeClearEvent(ext->pEvent);
```

Whenever a MON_DEACTIVATE is processed, the FSRFD will close the event as:

```
ZwClose(ext->eventHandle);
```

Kernel or Low Level Driver Interfaces

Every driver needs a `DriverEntry` routine that is called when the driver is first loaded. This routine for FSRFD is described in the Component Design section.

The FSRFD inserts "hook routines" that intercept relevant registry and file system calls. The Component Design section describes which hook routines are inserted and what they do.

Component Design

Global variables

pImonDevice

This global variable points to the device object created in the DriverEntry function for the "\\Device\\installmon" device.

DriverEntry

```
NTSTATUS DriverEntry(IN DriverObject, IN RegistryPath)
{
          IoCreateDevice for "\\Device\\installmon";
          pImonDevice = pDeviceObject returned above;
          ext = pImonDevice->DeviceExtension;
          ext->type = INSTALLMONINTERFACE;
          IoCreateSymbolicLink with
              "\\DosDevices\\installmon";
          for all IRP_MJ_* values upto
               IRP_MJ_MAXIMUM_FUNCTION {
            DriverObject->MajorFunction[IRP_MJ_*] =
                 ImonDispatch
          }
          Setup the unload driver function
          DriverObject->FastIoDispatch = address of
            our fast io dispatch table;
            Note that we are interested only in
            FAST_IO_*WRITE* routines for getting our
            entries, however we need to implement all
            of them to call lower layered drivers. All
            our FastIo funcs are called ImonFastIo*;
          Create the necessary mutexes;
          Use PsSetCreateProcessNotifyRoutine to set a
          process create callback routine
          ImonProcessCallback;
}
```

ImonProcessCallback

```
{
          // similar to ImonProcessCallback
          // This function is called every time a
```

```
            // process on the system is created or
            // deleted. We need to figure out (by
            // checking the parent id in our list)
            // if we need to add or remove this process
            // from our list
            lock the mutex for ProcessIdList
            if not activated just return;
            if (this is process create) {
               Look up the parent process id in the
               ProcessIdList
               If present, add this process id to the
                  ProcessIdList
            }
            else {
               if this process id is present then
                  remove the process id
            }
            release the mutex
    }
```

ImonDispatch

```
    {
            // similar to FilemonDispatch
            // Instead of registering a different
            // function for each IRP_MJ_* this function
            // is called for all of them and this one
            // dispatches the right on based on the
            // control code
            This gets called for all IRP_MJ_*;
            if the device extension type tells us
               INSTALLMONINTERFACE
                  call ImonDeviceFunc
            else
                  call ImonHookFunc
    }
```

ImonDeviceFunc

```
    {
            // similar to FilemonDeviceRoutine
            // This function is called whenever an Ioctl
            // comes from the Installmon process that is
            // meant to be a command for this FSRFD.
            This is a request from the INSTALLMON using
            the INSTALLMON device. We would mainly be
            processing IRP_MJ_DEVICE_CONTROL, although
            ImonFastIoDeviceControl should have been
            called. So if we come here just call
```

```
        ImonFastIoDeviceControl
        Call IoCompleteRequest?
}
```

ImonHookFunc

```
{
        // similar to FilemonHookRoutine
        // This is the hook function that is called
        // for all the I/O requests that is made by
        // the I/O manager that need to go through
        // this driver (i.e. for those requests
        // that we are filtering).
        We are interested in recording:
           IRP_MJ_CREATE where the Irp->
              Parameters.Create.Options indicates
              a new file create (as opposed to an
              existing file open)
           IRP_MJ_WRITE
        Note that we have to get the current
        process-id using PsGetCurrentProcessId and
        look it up in the ProcessIdList (note: you
        have to exclude the first process-id since
        that belongs to the Installmon and not the
        setup program) and only if that search is
        successful, record the entry
        Note that we need to get the filename from
           the FileObject using code similar to
           FilemonGetFullpath
        Also we should be recording the entry when
        the request is successfully completed. So do
        this in the completion routine in a manner
        similar to filemon.
        To record:
           create a relevant IMON_ENTRY record;
           Call ImonAddEntry with this record;
        Pass all Irps to lower layer driver using
           IoCallDriver and getting the lower layer
           device object from this device's
           ext->deviceObject
}
```

ImonFastIoDeviceControl

```
{
        // similar to both
        // FilemonFastIoDeviceControl and
        // ImonDispatchIoctl
        // This function gets called for all the
```

```
            // IOCTLs. If it is from Installmon, we need
            // to process the MON_* commands or else
            // just pass on the command to the lower
            // layer driver.
            Get the current device's extension
            if type indicates INSTALLMONINTERFACE {
               switch (IoControlCode) {
                  case MON_ACTIVATE:
                     call ImonActivate;
                     break;
                  case MON_DEACTIVATE:
                     Call ImonDeactivate;
                     break;
                  case MON_GETENTRY:
                     Call ImonGetEntry;
                     break;
               }
            }
            else {
               pass it down using deviceObject and the
               fastio hook for Ioctl
            }
   }
```

ImonAddEntry

```
   {
            // similar to ImonEnqueueRegEntry and
            // createRegEntry etc.
            // This function adds a IMON_ENTRY node
            // to our list: this list is eventually
            // returned to InstallMon
            create an entry rec from nonPagedPool
            Grab a mutex to modify EntryList
            Add the entry rec to EntryList
            release the mutex
            KeSetEvent for IMON event object
   }
```

ImonActivate

```
   {
            // similar to ImonActivate and various
            // Filemon funcs called by
            // FilemonFastIoDeviceControl
            // This is called by our own func that
            // handles IOCTLs when a MON_ACTIVATE is
            // sent by InstallMon
            Look at the ProcessIdList to ensure that we
```

```
              are not already active (1st process id).
              If we are, return with an error with that
              process id
              Grab a mutex
              Add a node to ProcessIdList with the
              processId;
              Release the mutex;
              Create the IMON event object;
              HookDrive(sysDrive);
              HookDrive(destDrive);
              HookRegistry();
    }
```

ImonDeactivate

```
    {
              // Same as above except for MON_DEACTIVATE
              If we are already deactivated
                 return with an error;
              If EntryList is not empty and this is not
                 a forced deactivation then
                 return with error;
              Clear and Delete the IMON event object;
              Free the ProcessIdList;
              Free the EntryList;
              UnhookRegistry();
    }
```

ImonGetEntry

```
    {
              // When the InstallMon sends a MON_GET_ENTRY
              // this function is called.
              Grab the mutex to access EntryList
              get next entry and remove from the
              list;
              if no entry then {
                 if (deactivated and first process in
                      the processIdList is dead) {
                    Make a new entry of "end of data"
                       type;
                 }
                 else {
                    there is some problem (should not
                       happen)
                 }
              }
              copy the entry into the OUT buffer
              release the mutex;
```

```
}
```

HookDrive, UnhookDrive

```
{
        // very similar to Filemon's HookDrive
        // Hence not described here
        // Similarly UnhookDrive
        // When a MON_ACTIVATE comes, we need to
        // make sure that all the requests to the
        // system drive or dest drive are filtered
        // through us. e.g. If "C:" is the system
        // drive, HookDrive will register this
        // as the filter driver for all IO for "C:"
        // Similary for UnhookDrive.
}
```

HookRegistry, UnhookRegistry

```
{
        //similar to Installmon's (Un)HookRegistry
        // We need to make sure that all Registry
        // functions are replaced, with our funcs
        // so that these funcs get called whenever
        // a process is trying to access the
        // registry. Our hook funcs will in turn
        // call the real funcs after queueing an
        // entry
}
```

ImonFastIo* routines

```
{
        // These are very similar to FilemonFastIo
        // routines except that we need to intercept
        // only FAST_IO_WRITE,
        // FAST_IO_MDL_WRITE_COMPLETE,
        // FAST_IO_WRITE_COMPRESSED,
        // FAST_IO_WRITE_COMPLETE_COMPRESSED
        Call the lower layer driver;
        if the request was successfully completed {
          Record the details into our entry rec
          and enqueue it similar to how ImonHookFunc
          does it;
        }
}
```

Interesting issues to deal with:

- Make sure that we use non-paged memory as required. e.g. all the nodes of the processIdList and entryList will be from non-paged pool. According to Bob, we do not need to always allocate non-paged memory – for Registry related nodes (i.e. when it is 'R' type IMON_ENTRY node) we can allocate paged memory. However this needs to be checked – since there will be pointers between these nodes, we need to make sure that this will work properly.
- We have to make sure that a 2-phase install works with this: some setups ask you to reboot the machine and after the reboot the setup continues. For the FSRFD we need to make sure that after the reboot the FSRFD is already loaded before the 2nd phase of the install starts. In the FSRFD we may need to make sure that when the "Runonce" registry key is updated (the installer is trying to do a 2-phase install and setting the 2nd phase exe as the value of "Runonce") we capure that info and accordingly co-ordinate with the Installmon to do the right thing. To ensure that this driver is started at the right time on start-up (since the Builder machine is going to be an in-house dedicated machine, it is okay), we need to add to the System registry (under `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`) appropriate values for `Start`, `Group` and `Tag` (and possibly others) value names. Actually this is going to be implemented in InstallMon as a user initiated event in which case the user informs the Builder UI (which in turn informs the InstallMon) that a reboot is imminent. In that case the InstallMon can try to find all possible ways in which the setup.exe is achieving this:
  - The "RunOnce" key or one of its other incarnations (e.g. RunOnceService, RunOnceEx etc) has been modified. We need to figure out exactly which one of the "*RunOnce*" can be modified.
  - The setup.exe has actually added itself (or another exe) to the startup folder. If that is the case, we can do the same trick here: replace that entry with an entry pointing to the BuilderUI with the original setup.exe value as an argument to the BuilderUI.
- An issue that hasn't been resolved is any user interaction (and user input) that has taken place during installation that is being monitored. For example, an installation may ask for a port number that it may store in a registry key. The solution suggested is as follows: This has to be a manual process. The Builder user should record all the manual interaction and manual data input that has taken place. He should recreate the same interaction in the custom DLL that the appInstallBlock provides for that app. This custom DLL at eStream app subscribe time can do the same thing that was observed during original application install. Alternatively we can request the ISV's co-operation in doing this. (May be this bullet should be transferred to a different doc such as the InstallMon LLD.).
- There is another unresolved issue about handling h/w or s/w dependent things that the installer does: we will have to handle this case by case basis and any knowledge we gain as a result of this, we should consolidate in the Builder components. e.g. we may notice that some installations may depend on IE4 or IE5 being there. Of course, one of the pre-requisites of the Builder is that it will be run on a pristine machines, so that we capture a maximal installation when it is taking place.

Testing design

Unit testing plans

This will be unit tested using the INSTALLMON program (or its early prototype). The INSTALLMON program sends all the required Ioctls like MON_ACTIVATE, MON_DEACTIVATE and MON_GETENTRY. The INSTALLMON output will be used to check the correctness of the FSRFD. This means the INSTALLMON itself should be assumed to be correct.

In addition to the above, we actually need to write one or more test programs that exercise the FSRFD. These test programs will be run as if they were App Installers i.e. as child processes of the INSTALLMON. The test programs should be written to exercise:

- All file systems (FAT, FAT32, NTFS, HPFS, compressed drives and others), since the FastIo routines need to be tested.
- For each of the file systems:
    - File create (with and without the old file being there)
    - File update (existing file appended as well as modified in the middle)
    - File touch (e.g. get the version of a DLL file) – this is not yet covered by this design
- Registry updates:
    - create a key
    - delete a subkey
    - delete a named value from a key
    - RegReplaceKey (we need to investigate if this is broken down into smaller reg calls).
    - RegRestoreKey (same comment as above applies)
    - RegSetKeySecurity (we failed to address this in the design)
    - RegSetValueEx
    - RegLoadKey and RegUnLoadKey

Stress testing plans

The FSRFD will be stress-tested using the above testing strategy by varying the rate at which files/registry are updated and under a variety of conditions (memory/disk, other processes).

Coverage testing plans

The unit testing above also covers Coverage testing.

Cross-component testing plans

This will be tested with the Installmon program which is enough for cross-component testing.

Upgrading/Supportability/Deployment design

Deployment: This will be used in-house, so no deployment considerations.

Open Issues

These issues are not necessarily FSRFD related, but are listed here just to remind ourselves.

- Do we need to worry about environment variables? It looks like most installations (and their apps) won't be looking at environment variables.
- What about the .lnk files (shortcuts). It looks like the client guys will have to change all the .lnk files based on the actual client's settings. This issue has been addressed either in the InstallMon or Packager. Pls see those documents. (The `IShellLink` interface has been suggested).
- Also what about when device drivers are installed? There is no impact on the builder but the client will need to reboot and hopefully the existing appInstall-Block and the custom initialization dll should be able to take care of it.

eStream Installmon Low Level Design

Sanjay M Pujare
<Date>

Functionality

The Installmon is a part of the Builder module that talks to the FSRFD to monitor file system and registry updates initiated by the Builder process. The FSRFD driver just intercepts such requests and records them and returns the recorded data to Installmon when requested by the latter. All the intelligence, such as any decision-making logic, resides in the Installmon.

Note:

1. **This does not cover those rare cases where an existing file is updated by an application install, and the eStream client would need to make the same updates. This kind of functionality is difficult to implement and will not be considered for 1.0.**

2. **Somewhere in the docs (may be the user docs?) it should be mentioned that the Builder's (or rather Installmon's) job could be made easier by the user by following these guidelines:**

    - **Make sure that values in all the registry keys are as distinct as possible. We may need to create a special Win2K or WinNT installation where each key (or ValueName within a key) will be created with a distinct or unique value as far as possible. E.g. If the default Windows installation creates 2 keys FOO and BAR and stores the same value "C:\Windows\System32" in both of them, we wouldn't know which one of those keys is used when a file is copied to "C:\Windows\System32". To solve this problem, all the effort should be made to ensure that FOO and BAR have distinct values.**

    - **When the Builder operator is installing an app under the Installmon, she should also make sure that the install script is given a distinct or unique value for each of the user inputs that may be used to set a registry key or an environment variable. This is especially necessary when the inputs seem to be totally unrelated. For example, vendor name and installation directory name. If both are entered as "Microsoft" then that could cause confusion to the Installmon.**

3. **There are 2 ways in which registry changes and file system changes can be captured:**

    - **using a kernel mode driver such as the FSRFD, Or**

    - **using a diff'ing mechanism for both the registry as well as the file system.**

In this design, we use both the approaches, and record any differences due to these two approaches. We give the user a chance to manually edit the registry/file changes.

Data type definitions

Interface definitions

All of the interfaces used to communicate with the FSRFD are described in the FSRFD LLD document. This LLD document will cover only interfaces with the other modules.

Interfaces with the Builder UI

The public methods of the InstallMon class described below are interfaces to the Builder UI. Specifically these are:

```
void InstallMon::startCapture(PUNICODE_STRING setup_exe,
  PUNICODE_STRING dest, bool upg, FileTable_t *fTbl);
```

This function is called when the user chooses to start monitoring an install. This function is called after the user has entered all the necessary data such as the setup.exe path etc.

`void InstallMon::stopCapture()`

This is called in rare cases when the user wants to terminate a running install in abnormal cases.

`bool InstallMon::checkSetupStatus()`

This is used by the UI to check the status of the install: whether the file list is ready to be processed and the registry list is ready to be processed etc.

`void InstallMon::getRegistryList()`

This function is used to finally get the list of registry changes that occurred after the install has taken place. This function allows the user to edit the list to manually add/delete/modify entries to be able to override.

`void InstallMon::getFilesList()`

This function is used to finally get the list of file system changes that occurred after the install has taken place. This function allows the user to edit the list to manually add/delete/modify entries to be able to override.

`void InstallMon::machineToBeRebooted()`

When the app install program is asking the user to reboot the machine (in the middle of the install) to continue installation, the user has to inform the Builder UI that a reboot is imminent. The InstallMon does the necessary bookkeeping to make sure that monitoring continues after the reboot.

```
void InstallMon::startCaptureAfterReboot(setup_exe, dest,
upg)
```

When the installation continues after the reboot, the Builder UI is automatically invoked, and it calls this function to inform the InstallMon to continue monitoring.

Access DB interface

We will be using an Access DB (or alternately the MSI databases as suggested by Bhaven) to store intermediate results of the Installmon process. There will be 2 tables used: Registry and Files.

Registry table

```
Fullpath :string;         // this is the full path of the key
ValueName :string;        // this is the value name: for
             // (default) use NULL
UpdateType: char;         // Add/update or delete, also
             // 'R' for removed
// combined (Fullpath, ValueName) should be unique i.e.
// primary key
```

Files table

```
Fullpath : string;
UpdateType : char;        // add, update
Kind: char;               // 'C'opied, 'S'poofed, 'E'stream
FileId: Number;
// Combined (Fullpath, UpdateType, FileId) should be unique
```

RegistryDiff table

```
Fullpath :string;
ValueName:string;
ValueType: char;
Value: something that can store all REG_* types
status: char;    // 'O'riginal, 'C'(add/change), 'D'
// combined (Fullpath, ValueName) should be unique
```

FilesDiff table

```
Fullpath: string;
Type: char;     // file 'F' or dir 'D' etc
Status: char;     // 'O'riginal, ...'C', 'A', 'D' etc
// (Fullpath
```

Component Design

```
struct FileTable_t {
  PUNICODE_STRING name;
  Char type; // spoofed, copied or eFS
  Int  fileId;
};

class InstallMon {
```

```
PUNICODE_STRING setup, destDrive;
bool interruptThread;
EventObject signalMonitor, signalSetup;
bool rebootReq = false;
bool afterReboot = false;
bool completed;
int exitCode;
bool upgrade;
someType envVars;
// TODO: combine the above 2 into an enum
int upgradeFileIdBegin = -1;
int currFileId;
static threadSetup(InstallMon iMon) {
  Remove all the environment variables except
  the ones in iMon->envVars;
  Start the Setup program;
  And wait for it to finish;
  iMon->exitCode = exit code from the process;
  when finished signal the signalSetup event;
}
static threadMonitor(InstallMon *iMon) {
     // this is the func that runs as a separate
     // thread, until we are interrupted or we
     // notice that the setup process has exited.
  get the current
  process id, system drive (using
  GetSystemWindowsDirectory), destDrive and send
  the MON_ACTIVATE message>
  Start threadSetup thread with setup_exe;
  processEnded = false;
  while (!interruptThread) {
       if (rebootReq) {
         only poll for Installmonevent;
         if (signal not set) {
           break from the while loop;
         }
       }
       else {
         WaitForMultipleObjects -> signalMonitor,
         Installmonevent and signalSetup;
       }
       switch (signal) {
       case Installmon event:
         get the MonitorEntry_t record;
         switch (regOrFile) {
         case 'R':
           switch (updateType) {
```

```
            case 'A':
            case 'U':
               add or update (KeyName, ValueName,
                  'N') to
                 registry table;
               break;
            case 'D':
               add (KeyName, ValueName, 'D') to
                 registry table;
         }
         break;
      case 'F':
         switch (updateType) {
            case 'A':
               // file creation
               add (fullpath, 'A', '?',
                 iMon->currFileId++) to files
                 table;
               break;
            case 'U':
               add (fullpath, 'U') to files
                 table;
               break;
         }
         break;
      case 'E':
         no more data to add;
         if (!processEnded) {
            // something wrong!!!!
         }
         break out of the while loop
         break;
   }
   break;
case setupevent:
   processEnded = true;
   Send MON_DEACTIVATE to the FSRFD;
   break;
case signalMonitor event:
   if (rebootReq) {
      kill the threadSetup thread
      clear the signalMonitor event;
   }
   else {
      // TBD?
   }
   break;
```

```
        } // switch
    } // while
    if (processEnded) {
        // normal setup process completion
        // registry and files tables should have
        // all the captured data from the FSRFD
        iMon->diffCapture();
        iMon->completed = true;
    }
} // threadMonitor
void commonCapture(setup_exe, dest) {
    setup = setup_exe;
    destDrive = dest;
    Start the threadFunc thread, and pass 'this' to
    it;
}
void diffCapture() {
  // Bhaven suggested that we could instead do an export
  // from regedit and do a text diff for registry diffs.
  // similarly: It seems the regular Unix diff tool also
  // compares directories. I don't know if it is
  // recursive. If it is, we might be able to use it.
  // Further investigation is recommended for doing
  // files diff.
  Query the OTI\BUILDERSTART key and get the value
  in builderStartTime and delete the
  OTI\BUILDERSTART key;
  // process the registry
  // step 1: query RegistryDiff table
  for each row in RegistryDiff table {
    query the corresponding key+valueName in the
    Windows registry
    if (exists) {
      if (no change to value) {
        update the row status to 'U' for unchanged
      }
      else {
        update the row status to 'C' (changed)
      }
    }
    else {
      update the row status to 'D' (deleted)
    }
  }
  // step 2: enumerate the whole Windows registry
  for each enumerated registry key+value {
    query the RegistryDiff table;
```

```
        if (a row exists) {
          if (status is 'U') {
            delete the row
          }
          else {
            status should be 'C' and values should be
            different bet table and actual registry
            or else display fatal error(?)
          }
        }
        else {
          // no previous value
          add a row to RegistryDiff with (fullpath,
            valueName, ValueType, Value, 'A') to mark
          it as an added registry key
        }
      }
      // now repopulate the FilesDiff table
      // step 1: query FilesDiff table
      for each row in FilesDiff table {
        query the corresponding Fullpath in the actual
        file system;
        if (exists) {
          if (no change (i.e. timestamp before
            builderStartTime)) {
            update the row status to 'U' for unchanged
          }
          else {
            update the row status to 'C' (changed)
          }
        }
        else {
          update the row status to 'D' (deleted)
        }
      }
      // step 2: traverse the whole file system
      for each file/dir in {systemDrive, destDrive} {
        query the FilesDiff table;
        if (a row exists) {
          if (status is 'U') {
            delete the row
          }
          else {
            status should be 'C' and the file/dir
            timestamp should be after builderStartTime
            or else display fatal error(?)
          }
```

```
        }
        else {
           // no previous value
           add a row to FilesDiff with (fullpath,
              'F' or 'D', 'A') to mark it as an added
           file/dir;
        }
     }
  }
  void recursiveFileGetter(curDir) {
     add a row to FilesDiff as (curDir, 'D', 'O');
     for (each element of curDir) {
        if (element is a dir) {
           recursiveFileGetter(element);
        }
        else { // has to be a file(?)
           add a row to FilesDiff as (element, 'F',
                'O');
        }
     }
  }
public:
  InstallMon(?);
  void setEnvVars() {
     allow the user to set environment variables
     in an interactive way;
     Get the values in envVars;
  }
  void startCapture(PUNICODE_STRING setup_exe,
        PUNICODE_STRING dest, bool upg, FileTable_t *fTbl) {
     clear the Access DB registry, files,
        registryDiff and FilesDiff tables;
     if (upg) {
        populate the files table with data from the
           fTbl array;
        upgradeFileIdBegin = last file id + 1;
        currFileId = upgradeFileIdBegin;
     }
     else {
        currFileId = 0 or 1 (depends on where to start);
     }
     Query the whole registry and
     for (each key and valueName pair) {
        add (key, valueName, valueType, value, 'O') to
           registryDiff;
     }
     Get the current time and store it in some format
```

```
    in a registry key OTI\BUILDERSTART;
    for (curDrive in {systemDrive, dest}) do
    {
      Root = root of curDrive (e.g. "C:\\");
      recursiveFileGetter(Root);
    }
    interruptThread = false;
    afterReboot = false;
    commonCapture(setup_exe, dest, upg);
}
void stopCapture() {
      // abnormal capture termination
      // TBD
      // also there might be normal cases where this
      // func is called when the setup process exit
      // is not detected for some reason (or doesn't
      // happen)! May be we don't have to worry about
      // these things.
}
bool checkSetupStatus() {
   if (!completed) {
      display error and return false;
   }
   if (exitCode is not okay) {
      display error and return false;
   }
}
void getRegistryList() {
   checkSetupStatus() and conditionally return;
   Using appropriate SQL, show (fullpath, ValueName)
     tuples that exist in Registry but not in
     RegistryDiff;
   Tell the user that these were captured by FSRFD
   (installmon) driver but not by the diff process;
   If user chooses to remove any tuples from the
   shown list, mark these with status as 'R' in the
   Registry table;
   Using appropriate SQL, show (fullpath, ValueName)
     tuples that exist in RegistryDiff but not in
     Registry;
   Tell the user that these were captured by diff
   but not by the FSRFD (installmon) driver;
   If user chooses to add any tuples from the
   shown list, add these tuples to the Registry
   table with status copied from RegistryDiff;
   Remove all the rows from the Registry table
   where status is 'R';
```

```
    Now we have all the correct entries in Registry;
    Read each row of Registry and into an array to be
    passed to the caller (most probably the caller of this
    func would have passed an array in which we should
    pass these rows);
  }
  void getFilesList() {
    checkSetupStatus() and conditionally return;
    Using appropriate SQL, show (fullpath)
      tuples that exist in Files but not in
      FilesDiff;
    Tell the user that these were captured by FSRFD
    (installmon) driver but not by the diff process;
    If user chooses to remove any tuples from the
    shown list, mark these with status as 'R' in the
    Files table;
    Using appropriate SQL, show (fullpath)
      tuples that exist in FilesDiff but not in
      Files;
    Tell the user that these were captured by diff
    but not by the FSRFD (installmon) driver;
    If user chooses to add any tuples from the
    shown list, add these tuples to the Files
    table with status copied from FilesDiff;
    Remove all the rows from the Files table
    where status is 'R';
    Now we have all the correct entries in Files;

    Using appropriate SQL, select files (and not
    dirs) where the file has only 'U' and not 'A':
    this means the setup modified an existing file
    and we cannot handle this; so if this happens
    show a fatal error;

    Read each row of Files and classify it into 'C',
    'S' or 'E' based on the following logic:
    If the file belongs to the app install directory
    (or app drive?) it is an 'E';
    If the file is smaller than a threshold then
    it is 'C' or else it is 'S';

    if (upg) {
      we need to create new file-ids for directories
      that contain new files or directories
      int prevStart = upgradeFileIdBegin;
      int prevEnd = currFileId;
      while ((prevEnd - 1) > prevStart) {
```

```
      using appropriate SQL, select rows from the files
      table where fileId >= prevStart and kind is not
         'C';
      for each such row {
         fullpath = fullpath in the row; (e.g. "C:\A\B")
         dir = parent directory of fullpath (e.g.
               "C:\A")
         select in files a row where fullpath = dir and
         fileId >= upgradeFileIdBegin;
         if (doesn't exist) {
            add in Files a row with (dir, 'X', '?',
               currFileId++);
            // Here 'X' stands for "new file id for a
            // directory created because one of the
            // children has a new file id
         }
      }
      prevStart = prevEnd;
      prevEnd = currFileId;
   }
}
```

Also for each file, change the absolute path to an envVar relative or registry-value relative one: this is a non-trivial task. The way to do this is proposed below. However we need to refine this algorithm based on actual Builder runs on real apps:

For a basic Win2K (or WinNT as the case may be) system create a list of registry keys (and env vars) whose values are normally used as destination paths for copying various kinds of files. To this list also add this app's dest dir as one of the values. Also add all the registry keys/values added as part of this app's install. Create an access table that looks like:

```
Key: String;     // registry or env or other name
Type: char;      // 'R'egistry, 'E'nv, 'D'est dir etc.
Source: char;    // 'S'ystem, 'A'pp, 'U'nknown?
Value: string;
```

Now all of the 'E' files should be relative to dest dir (i.e. 'D' type above). Also any sub-directory strings should either be hard-coded or based on some registry key values where the registry key is of source 'A' (app) above.

```
    All the 'C' and 'S' files should be relative to one of
    the registry keys of source 'S' above and preferably
    type 'R' (since environment variables are not used un-
    der Windows?).

    If there is a sacred file set (i.e. files that are
    so "sacred" for eStream apps, that they were installed
    on the client when the eStream client was installed),
    we need to remove those files from this list;

    Set these values in the table and read the whole
    table in an array (most probably the caller of
    this func would have passed an array in which we
    should pass these rows);
}
void machineToBeRebooted() {
     //The setup is going to reboot the machine.
     //Note that there is a race condition possible here.
     // Suppose the setup.exe has displayed a dialog
     // asking the user to confirm a reboot. At this time
     // the setup.exe has still not written to "Runonce".
     // Only after the user has confirmed (by pressing
     // Okay) will the program write to "runonce" and
     // auto reboot. In that case, this function will be
     // called at the wrong time. We may need to modify
     // the FSRFD to detect changes to "Runonce".
     rebootReq = true;
     signal the signalMonitor event;
     wait for the threadMonitor thread to end;
     Query either our Access database or the actual
     registry to see if Runonce key is set/updated.
     if (not) {
       we cannot handle this reboot situation;
       give fatal error;
       // may be we can look at the
       // Start\Programs\Starup folder and do the
       // same thing we did with Runonce key
     }
     OldRunonce = Old value of Runonce key;
     Set the new value of Runonce key as our Builder
     name (or whichever EXE has the installmon code)
     followed by OldRunonce as the argument to that
     EXE and the destDrive as the next argument and
     upgrade as the next arg;
     Prompt the user to go ahead and say Okay to
     Setup's dialog warning that it is going to
     reboot;
```

```
    }
    void startCaptureAfterReboot(setup_exe, dest, upg) {
        afterReboot = true;
        commonCapture(setup_exe, dest, upg);
    }
};
```

Interesting issues to deal with:

Testing design

Unit testing plans

The unit testing of installmon will be done in conjunction with the FSRFD unit testing. This has been described in the FSRFD-LLD document. In addition, we will be using the sysdiff tool to validate the results of the Installmon.

Stress testing plans

All of the 14 or so applications (that OTI is planning to convert to eStream) installs will be tested under the installmon. Any issues found will be used to fix and improve the installmon functionality.

Coverage testing plans

The testing of the Builder/installmon on the 14 or so app installs should give us enough coverage. Considering that Builder/installmon will be used in-house for some time makes some of the testing issues less significant.

Cross-component testing plans

Will be tested as a component of the whole builder.

Upgrading/Supportability/Deployment design

Deployment: This will be used in-house, so no deployment considerations.

Open Issues

- On one of the newgroups someone mentioned a key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs] that we can look at for the shared DLLs that cannot be installed. Need to investigate.
- 2-phase install: some installs need a reboot after which they continue. The key to look at is Runonce (under the same path as SharedDLLs I think).

eStream 1.0 Client

Registry Spoofing Database

Omnishift Technologies, Inc.
Company Confidential

Functionality

This component is a device driver that is started and stopped by the eStreamClient.exe program.

The registry-spoofing database is a list of registry entries that the registry spoofer redirects whenever client programs or the windows kernel requests spoofed registry keys. The reason that we do this is to present different views of the registry depending on when different users are logged into the system. Upon completion of User Login a startup program called eStreamControl is begun to initialize the following client components (not necessarily in this order).

1. Registry Spoofing
2. File Spoofing
3. Z File System Set up
4. eStream Logon and user Verification.

Registry spoofing is begun by a call into an eStream Device Driver from the eStreamControl program that will start the kernel level registry spoofer. When the user logs out the eStreamControl will make a call to the eStream Device Driver to shut down the registry spoofing. If another user logs on to the same computer who is not enabled as an eStream user then that users view of the registry is identical to what it would be if eStream were not installed on the client computer.

Data type definitions

The registry entries to be spoofed are provided on a per-user set. Every user who logs on to a computer with eStream installed will have their own set of spoofed registry entries. These registry entries will be stored under HKEY_CURRENT_USER. This will enable simple back up of the keys when the user logs out. The HKEY_CURRENT_USER Keys are automatically backed up to HKEY_USERS when the user logs out and restored when the user Logs back in.

Keys that need to be spoofed are always in one of the following places

HKEY_CLASSES_ROOT (HKCR)
HKEY_LOCAL_MACHINE (HKLM)
HKEY_CURRENT_CONFIG (HKCC)

Spoofed Keys are stored in the registry database under HKEY_CURRENT_USER. This portion of the registry database is set up when a user logs on and is stored back into the HKEY_USERS when the user logs off.

The App install block uses a .reg file. The keys in the .reg file are added to the registry by the regedit program provides.

The structure of the keys will be as follows.

```
HKEY_CURRENT_USER
        Estream
                Registry Spoof
                        Add
                                HKCR
                                        ...
                                HKLM
                                        ...
                                HKCC
                                        ...
                        Remove
                                HKCR
                                        ...
                                HKLM
                                        ...
                                HKCC
                                        ...
```

The eStream Registry spoofer needs to be able to delete keys as well as overwrite keys and add keys. The Overwrite and create key functions are provided by the Add Sub Keys. The Remove key may be a very sparse tree, but some software units will probably need the facility.

Components

Spoof Device Driver

This is a device driver that has a DEVICE_IOCTL command to toggle registry spoofing on and off.

Spoof Regedit Utility Program

The utility program Regedit is not aware that registry spoofing is going on. It would be very helpful if an enhanced version of Regedit was developed that will be aware of registry spoofing and will identify keys that are added, deleted, or modified by the registry spoofing.

eStreamControl

This is an executable program that is started when the user logs on. The function of this program is to start up the eStream services that are required by the user to run eStream Applications. This program also shuts down eStream application services when the user logs out.

Interface definitions

Registry Spoofing uses the same interfaces that the registry uses. The only interface that is required is a device driver call to toggle registry spoofing on and off. The actual registry spoofer in the Windows Kernel will go to HKEY_CURRENT_USER\eStream to find its spoof database.

StartRegSpoofing()

Causes the registry spoofer to begin spoofing using the current database.

StopRegSpoofing()

Component design

The registry spoofing device driver will intercept all of the following kernel level registry calls.

RtlCheckRegistryKey

```
NTSTATUS
  RtlCheckRegistryKey(
  IN ULONG  RelativeTo,
  IN PWSTR  Path
  );
```

RtlCheckRegistryKey checks for the existence of a given named key in the registry.

RtlCreateRegistryKey

```
NTSTATUS
  RtlCreateRegistryKey(
  IN ULONG  RelativeTo,
  IN PWSTR  Path
  );
```

RtlCreateRegistryKey adds a key object in the registry along a given relative path.

RtlWriteRegistryValue

```
NTSTATUS
  RtlWriteRegistryValue(
  IN ULONG  RelativeTo,
  IN PCWSTR  Path,
  IN PCWSTR  ValueName,
  IN ULONG  ValueType,
  IN PVOID  ValueData,
  IN ULONG  ValueLength
  );
```

RtlWriteRegistryValue writes caller-supplied data into the registry along the specified relative path at the given value name.

RtlDeleteRegistryValue

```
NTSTATUS
  RtlDeleteRegistryValue(
  IN ULONG  RelativeTo,
  IN PCWSTR  Path,
  IN PCWSTR  ValueName
  );
```

RtlDeleteRegistryValue removes the specified entry name and the associated values from the registry along the given relative path.

```
NTSTATUS
  ZwCreateKey(
  OUT PHANDLE  KeyHandle,
  IN ACCESS_MASK  DesiredAccess,
  IN POBJECT_ATTRIBUTES  ObjectAttributes,
  IN ULONG  TitleIndex,
  IN PUNICODE_STRING  Class  OPTIONAL,
  IN ULONG  CreateOptions,
  OUT PULONG  Disposition  OPTIONAL
  );
```

Path

Specifies the registry path according to the *RelativeTo* value. If RTL_REGISTRY_HANDLE is set, *Path* is a handle to be used directly. If the path intercepts any spoofed registry path then the path will be redirected to HKEY_CURRENT_USER\eStream.

RtlQueryRegistryValues

```
NTSTATUS
  RtlQueryRegistryValues(
  IN ULONG  RelativeTo,
  IN PCWSTR  Path,
  IN PRTL_QUERY_REGISTRY_TABLE  QueryTable,
  IN PVOID  Context,
  IN PVOID  Environment  OPTIONAL
  );
```

RtlQueryRegistryValues allows the caller to query several values from the registry subtree with a single call.

All of these Kernel level Registry functions need to be intercepted. If the path variable intercepts any path inside HKEY_CURRENT_USER\estream then the call will be redirected to that key.

Private Helper Function

BOOL IsPathSpoofed([in] PCWSTR Path, [out] PCWSTR eStreamPath);

This will return TRUE if the path intercepts an eStream Key, FALSE otherwise. If the function returns TRUE then the corrected Spoof path will be placed in the eStreamPath argument.

Re-Registration of Objects

Many application programs such as Office Applications will register themselves every time they execute on the client machine. The Registry Spoofer will need to intercept these writes and re-direct them to the HKEY_CURRENT_USER\eStream set of registry keys.

Testing design

Generating Reg Key files and then using the normal regedit program to determine if the keys are being returned correctly will test this unit. Adding and removing registry keys using a client program that checks the spoofed path to the key can test this component quickly.

The registry-spoofing diver will need to be tested with the following Windows SDK Registry Functions.

Registry Functions

The following functions are used with the registry.

Function	Description
RegCloseKey	Releases a handle to the specified registry key.
RegConnectRegistry	Establishes a connection to a predefined registry handle on another computer.
RegCreateKeyEx	Creates the specified registry key.
RegDeleteKey	Deletes a subkey.
RegDeleteValue	Removes a named value from the specified registry key.
RegDisablePredefinedCache	Disables the predefined registry handle table of **HKEY_CURRENT_USER** for the specified process.
RegEnumKeyEx	Enumerates subkeys of the specified open registry key.
RegEnumValue	Enumerates the values for the specified open registry key.
RegFlushKey	Writes all the attributes of the specified open registry key into the registry.
RegGetKeySecurity	Retrieves a copy of the security descriptor protecting the specified open registry key.
RegLoadKey	Creates a subkey under **HKEY_USERS** or **HKEY_LOCAL_MACHINE** and stores registration information from a specified file into that subkey.
RegNotifyChangeKeyValue	Notifies the caller about changes to the attributes or contents of a specified registry key.
RegOpenCurrentUser	Retrieves a handle to the **HKEY_CURRENT_USER** key for the user the current thread is impersonating.

RegOpenKeyEx	Opens the specified registry key.
RegOpenUserClassesRoot	Retrieves a handle to the **HKEY_CLASSES_ROOT** key for the specified user.
RegOverridePredefKey	Maps a predefined registry key to a specified registry key.
RegQueryInfoKey	Retrieves information about the specified registry key.
RegQueryMultipleValues	Retrieves the type and data for a list of value names associated with an open registry key.
RegQueryValueEx	Retrieves the type and data for a specified value name associated with an open registry key.
RegReplaceKey	Replaces the file backing a registry key and all its subkeys with another file.
RegRestoreKey	Reads the registry information in a specified file and copies it over the specified key.
RegSaveKey	Saves the specified key and all of its subkeys and values to a new file.
RegSetKeySecurity	Sets the security of an open registry key.
RegSetValueEx	Sets the data and type of a specified value under a registry key.
RegUnLoadKey	Unloads the specified registry key and its subkeys from the registry.

The following are the initialization-file functions. They retrieve information from and copy information to a system- or application-defined initialization file. These functions are provided only for compatibility with 16-bit versions of Windows. New applications should use the registry.

Function	Description
GetPrivateProfileInt	Retrieves an integer associated with a key in the specified section of an initialization file.
GetPrivateProfileSection	Retrieves all the keys and values for the specified section of an initialization file.
GetPrivateProfileSectionNames	Retrieves the names of all sections in an initialization file.
GetPrivateProfileString	Retrieves a string from the specified section in an initialization file.
GetPrivateProfileStruct	Retrieves the data associated with a key in the specified section of an initialization file.
GetProfileInt	Retrieves an integer from a key in the specified

	section of the Win.ini file.
GetProfileSection	Retrieves all the keys and values for the specified section of the Win.ini file.
GetProfileString	Retrieves the string associated with a key in the specified section of the Win.ini file.
WritePrivateProfileSection	Replaces the keys and values for the specified section in an initialization file.
WritePrivateProfileString	Copies a string into the specified section of an initialization file.
WritePrivateProfileStruct	Copies data into a key in the specified section of an initialization file.
WriteProfileSection	Replaces the contents of the specified section in the Win.ini file with specified keys and values.
WriteProfileString	Copies a string into the specified section of the Win.ini file.

Obsolete Functions

These functions are provided only for compatibility with 16-bit versions of Windows.

RegCreateKey
RegEnumKey
RegOpenKey
RegQueryValue
RegSetValue

Unit testing plans

Unit testing can be performed with a simple client program that reads and writes keys in Spoofed Key folders.

Stress testing plans

Normal Windows operation reads and writes keys in high volume. One possible stress test would be to load and unload keys from the HKEY_CURRENT_USER/eStream/ database while simultaneously reading and writing the corresponding sub key folders.

Coverage testing plans

Test adding and removing keys from each of the spoofed key sets.

Add a key to each of the following points

HKEY_CURRENT_USER/eStream/Add/HKCR
HKEY_CURRENT_USER/eStream/Add/HKLM
HKEY_CURRENT_USER/eStream/Add/HKCC

Read the keys back from their corresponding keys in the following points.

HKEY_CLASSES_ROOT
HKEY_LOCAL_MACHINE
HKEY_CURRENT_CONFIG

Key deletion will also need to be checked. To accomplish this a set of keys is added to each of the following key folders

HKEY_CLASSES_ROOT
HKEY_LOCAL_MACHINE
HKEY_CURRENT_CONFIG

Keys are added to the corresponding Key Folders

HKEY_CURRENT_USER/eStream/Remove/HKCR
HKEY_CURRENT_USER/eStream/ Remove /HKLM
HKEY_CURRENT_USER/eStream/ Remove /HKCC

A client application reads the keys using the windows SDK functions

RegLoadKey
RegOpenKey

Cross-component testing plans

The install manager needs to install key sets using a .reg file. Using the regedit program can check this to see if the keys have been installed correctly.

eStream 1.0 Client UI Modules

Omnishift Technologies, Inc.
Company Confidential

Requirements

Here is a list of the eStream Client UI requirements

1. Administration of the Application Subscription and installation process
2. Cache and File system Management
3. License Management
4. User Account Log in
5. Application Syncing

Users running the eStream system need a simple system to subscribe and unsubscribe applications. The user interface that provides for this will come through the web browser. A browser plug in ActiveX control can provide a simple way for ASP web pages to interact with the user client system.

The File system driver will be loaded automatically when the user logs on. The Cache manager provides efficient data streaming to the file system driver. The Cache Manager is a worker thread inside the Client UI program.

License management provides tokens that will allow a user to access specific application files from an eStream server. The License Manager is a set of threads running inside the eStream Client program that acquires, renews, and releases access tokens as a subscribed application runs.

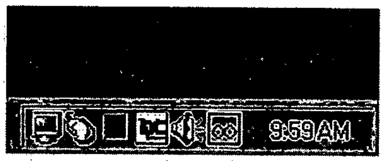
In order to access application executable and data files from an eStream Server a user must log onto an ASP server. The client UI program will provide this account management.

Users can access eStream from different computers at different times. Changes in Application subscription status synchronized between these different computers when the user logs on.

Functionality

The eStream Client UI module supports reporting eStream-specific error & informational messages to the client user and solicits replies when appropriate. It allows the eStream client user to view and change the list of applications currently installed on the client system and the list of ASP accounts currently known to the client system.

The eStream client will have a tray icon that will allow the user to access administration and control functions. Tray icons are small icons on the right side of the desktop toolbar. These tray icons give the user the indication that the eStream client is running and allow the user to access administrative functions that infrequently used.



eStream Tray Icon

Figure 1 Tray Icon

Right clicking on the tray icon produces a pop up menu that would allow the eStream user to access the following functions.

- Set the cache size and review cache performance and utilization
- Log the user into and out of ASP Accounts
- Set the proxy server IP address
- Set the ASP server
- Access the ASP web page and view accounting and utilization information.

Normal eStream The App Install manager will update Start Menus and desktop shortcuts in a manner identical to a native installation of the same application. The user does not interact with the eStream client when running subscribed applications.

Components

eStreamClient.exe

This program is located in the startup folder and will be run when the user logs onto a system with at least one eStream subscribed applications.

- Log on the ASP
- Start the file Spoofer
- Start the EFS file system
- Start the License Manager and the Cache Manager
- Get run tokens from the LRM
- Contact the ASP(s) and update changes in subscribed application sets

When the initialization is complete the eStreamClient.exe program will wait for the user to log off the system. When the user logs off the eStream device drivers (registry and file spoofers) are halted, LRM tokens and released, and the eStreamClient.exe Program exits.

Browser Plug In

An OLE Automation client is the simplest way for a web page to access internal configuration parameters on a Microsoft Windows Client. An OLE automation Client will allow a Browser script, VB Program, or VC program to communicate directly with the Client executable program.

For a browser Script the eStream Client executable could be controlled using an object tag.

Here is an example of HTML code that uses and Object Tag to connect to an OLE Automation server.

```
<OBJECT ID="ESTREAMCLIENT"

 CLASSID="CLSID:D8D77E03-712A-11CF-8C70-00006E127B7"

CODEBASE=http://www.aspcorp.com/eStreamClient.exe

DATA="InstallSetASPOffice1.ODS"?

</OBJECT>
```

Interfaces provided by the eStream Browser Plug In

AppInstall (String AppInstallBlockPATH)

The App install block contains a header section, variable section, and other blocks. The App Install block is large so a file link is sent to the AppInstall interface where the AppInstall block has been installed. This will probably be in some temporary folder.

The AppInstall function will need to build an Uninstall block that will list the components of the subscribed application. Uninstalling applications cleanly and reliably is as important as important as installing them. The behavior of the App install manager should mimic the Add/Remove panel from the Windows Control panel.

AppUninstall(AppId)

Uninstall the application. The header file of the app install block that comes with the application when it was originally subscribed provides the AppId.

XML data islands working together with Active Server Pages would provide a simple and reliable system for administrating Applications.

Application Database

A database containing application subscription information needs to be maintained on the client. This database can be displayed back to the user using XML tags. This database needs to contain enough information to provide the following functions the complete uninstall of subscribed installation, accounting information, and access to ASP accounts.

Design of the Client UI Program

The eStreamClient program provides all of the run time functionality that the client requires. The modules absorbed by this unit include

- Client Log On
- App Install Manager
- License Subscription Manager
- Client Network Manager
- Cache Manager

The App Install manager works in conjunction with the ASP web page to allow users to subscribe and un-subscribe applications. The coordination between the browser plug-in and the App Install Manager will probably require some kind of a COM interface into the eStreamClient program.

The License Subscription Manager manages access tokens. Access Tokens are generated by the eStream Server for a limited period of time and must be renewed when they expire. The eStream Client UI has a set of threads that deal with this process.

The Client Network Manager is the portal through which all communication between cache manager, license subscription manager, and the eStream server flows.

The Cache Manager together with the file system driver is the core component of the eStream product. These two components serve application executables over the network to mimic native installation of application programs.

eStream 1.0 eStream Client Network Component

Omnishift Technologies, Inc.
Company Confidential

Functionality

The client network component communicates with the following servers for the types of requests listed.

Account server

Validate a user for this ASP and get subscription information

DRM server

1. Validate a license for a subscribed app

App server

1. Get an app install block for a subscribed app
2. Open a file/directory for a subscribed app
3. Various file requests on a previously opened file/directory

Data type definitions

Definitions provided here are only for data types that are shared among components; this will take coordination with these other components. Definitions should be as C-language structures, regardless of how they'll be implemented.

Interface definitions

Interfaces

Only a few file operation interfaces are listed

ValidateUser()

Input:

- account server to query
- user data to send, including perhaps
  - ASP identifier

    - username
    - password
    - client certificate

Output:

- subscription info for this user, including perhaps
    - currently subscribed apps
    - serial number for each app
    - DRM server to use for validation for each app

ValidateLicense()

Input:

- DRM server to query
- subscription serial number to validate
- client certificate

Output:

- access token for this licensed subscription
- app server to use for data requests

GetInstallBlock()

Input:

- app server to query
- subscription serial number
- client certificate

Output:

- app install block for this subscribed app

AppOpenFile()

Input:

- app server to query
- access token (possibly NULL)
- file name

Output:

- handle to use for further file requests

AppReadFile()

Input:

- app server to query
- access token
- file handle
- buffer to fill
- offset
- length

Output:

- filled buffer
- number of bytes read

UploadAppProfileDataRequest()

Input:

- account server to upload to
- app profile data

Output:

- success/failure

Component design

Testing design

Unit testing plans

Stress testing plans

Coverage testing plans

Cross-component testing plans

eStream Registry Spoofer current status

Introduction

Discussions about the need for active registry spoofing using a kernel level device driver and a complex database are presented here. There is a strong desire by the client design group to simply the design of the eStream software.

Reasons for Registry Spoofing

1. Leaving the registry pristine
2. Allowing multiple users to have different views of the same computer

One of the design goals of eStream is to provide a system of supplying applications to a user while having a minimal impact on that users system. The normal process of installing application software on a computer will make a large number of changes to the registry database on the client computer. One possible advantage of registry spoofing is keeping the client computers registry unmodified as applications are subscribed and un-subscribed.

Another advantage of registry spoofing is to allow different users of the same computer to have access to different versions of software packages. This could be an advantage in situations where individuals share computers and eStream applications subscriptions are granted to individual users not to all the users of a particular computer. Two different users could use differing versions of the same application. A single user who does not subscribe to eStream could use one version and another user through an eStream application subscription could use a more current version of the same application.

Reasons for not doing Registry Spoofing

1. Increased complexity of the eStream client software
2. Difficulty of testing
3. Adding complexity to the app install process.

The difficulty of creating a kernel level registry spoofer is well understood by the members of the engineering team that have worked on Windows device drivers. The Registry is a component of the operating system that is critical to the proper functioning of both the Windows operating system kernel and most application software. It is accessed frequently by both the kernel and application software and any malfunction in its operation would have devastating consequences to the reliability of the eStream product. There are over 33 Windows SDK functions that access the registry directly and

probably hundreds more that access it indirectly. Testing all of these functions would be a very difficult undertaking and would be critical to the success of the eStream product.

In addition to the complexity added by a kernel level registry spoofer the addition of a set of spoofed registry entries for each subscribed application would add additional overhead, for both testing and installation, for each application that is available through the eStream subscription process.

Other Spoofing Options

1. Normal Installation of eStream Subscribed Apps
2. Logon time manipulation of the registry database

Possible alternatives to spoofing the registry database include keeping the registry entries of subscribed applications in a separate database that would be added to the registry when the eStream client is started, when the user logs on, or at system boot time. These registry entries would be removed at logout or shutdown time.

The popular consensus is to no perform registry spoofing at all on the eStream Client. The major advantage of this approach is to simplify the client application subscription process. The normal application install program that comes with each application could be used to subscribe an application if some way could be found to force the installer to place the application executables and portable data files on the EFS file system.

Conclusions

We have decided to abandon registry spoofing.

eStream 1.0 Installation Manager

Omnishift Technologies, Inc.
Company Confidential

Functionality

The Installation Manager consists of the following sub-components

Installshield

Installshield is the industry standard for building installation sets for Microsoft Windows. Installshield will take a set of executables and data files and create a media installation. The Installshield environment provides a scripting language that will allow a high degree of customization of target installation. The essentials issues for any installation are.

1. How much of the application does the user wish to install?
2. Is the users system capable of running the application?
3. Where does the user wish to install the application?
4. Does the user have enough space to install the applications?

Installshield has a wizard that will set up a project. When the install shield program is compiled a media must be specified. The most common media types are floppy, CD Rom, and Web media builds. For eStream we may have to ask the clients to reboot the machine since we are installing kernel mode components that might need a reboot to take effect.

Install From the Web

This program is another product that is sold by Installshield that will take a complete installation set and create a single executable .exe that can be easily downloaded from a web site.

Uninstaller

Installshield will provide an uninstaller when it builds the install program.

Registry Settings

There are three ways that Installshield can patch the system registry.

1. Run regsvr32.exe on self-registering .dll files. When the uninstaller is run it will use regsvr32.exe /u to un-register the .dll file.
2. Patch the registry statically.
3. Patch the registry based on Installation Options from the install shield script program.

Artwork

The Installshield program for eStream will require a splash screen and possibly one or two other artwork components.

Data type definitions

The data that the Installation manager uses

1. Device Drivers
2. COM Libararies
3. COM Executables
4. Registry files

Interface definitions

Testing design

Testing of the eStream Installer must take place on computers when Visual Studio has not ever been installed.

Unit testing plans

Install the eStream Client using the binary file installer from InstallShield. Since the number of installation options will be kept to a minimum this test should not take very long.

Installation of eStream Binaries from the InstallShield Installer is the first step of testing all new revisions of eStream.

Stress testing plans

The installation should be tested on a wide variety of computers with special emphasis on testing the installation on computers that do not have Visual Studio installed on them.

Coverage testing plans

Need component list for generating this test plan

Cross-component testing plans

All client components are installed with this piece so we don't have too much to worry about here.

eStream 1.0 License Subscription Manager (LSM)

Omnishift Technologies, Inc.
Company Confidential

Requirements

The purpose of the License Subscription manager is to acquire and manage list of tokens that will allow the cache manager to access application executable and data files on remote eStream servers.

Functionality

This component is a set of worker threads that are part of the Client UI process.

The LSM manages the users subscriptions to the different ASP accounts. It is part of the client component downloaded on a client machine. The LSM starts running when the client component starts running and is always active when the client component is running. Users on a given machine establish a connection with the ASP account servers from which they have subscribed applications. Users can add and delete the applications that are subscribed from the ASP accounts. The LSM makes the appropriate calls to the account servers to perform those actions. It gets serial numbers for the applications that are being subscribed and deletes them for the applications being un-subscribed (which are all part of the ASP ID Block). When the users start running any of the subscribed eStream applications, the eStream file system first queries the LSM before servicing any requests. The LSM in turn gets the appropriate access tokens from DRM servers along with the identities of application servers that can be used to run the applications. It uses the client identification (serial number) obtained when the connection to the ASP was made. At the same time, the LSM can decide to cache the access tokens and the identities of the application servers and decide to serve them directly from its cache. The eStream Cache Manager informs the LSM when applications start and end. The LSM keeps track of when access tokens are expiring and can request for additional access tokens when applications are running and the current one is about to expire.

Component design

The License Manger communicates with four other logical units inside the eStream Client.

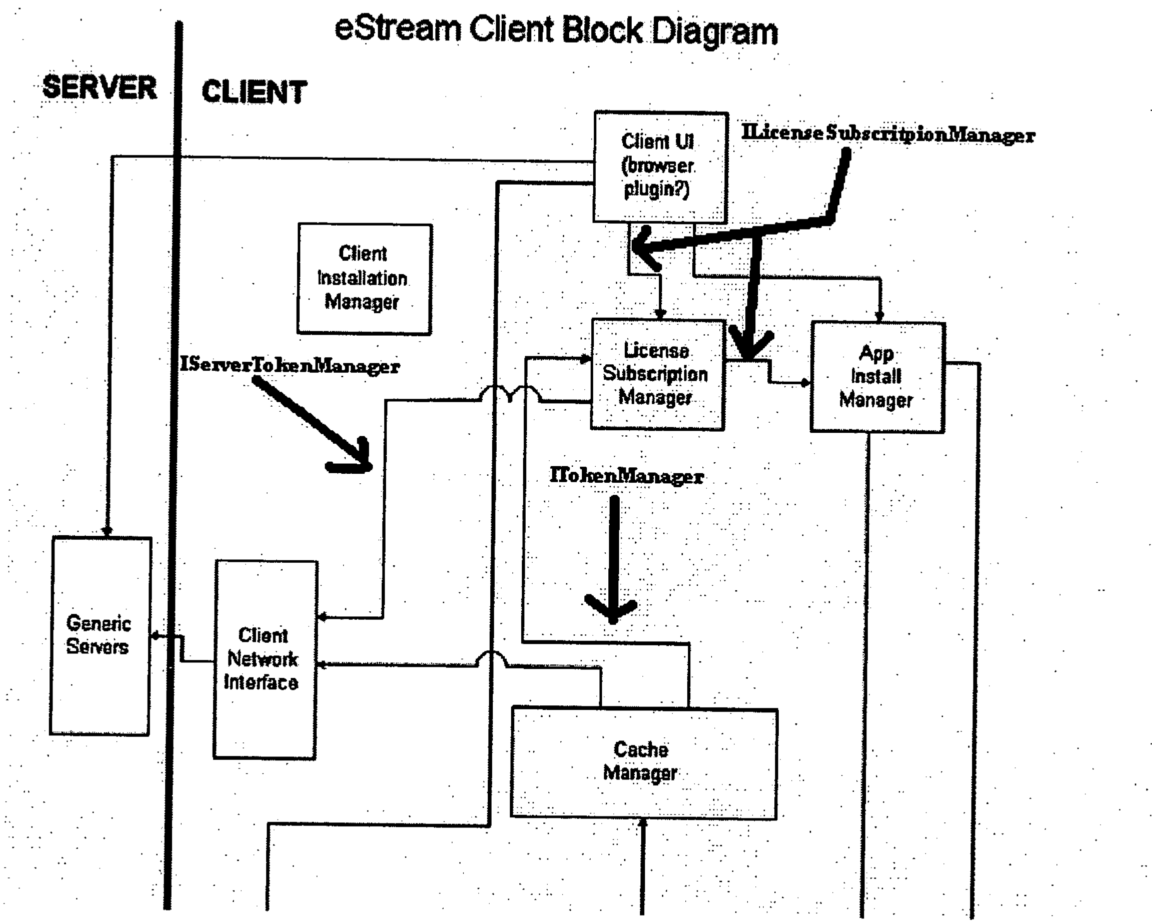


Figure 1 LSM interfaces

This component may be a COM server component. We may decide to implement some of the functions of this unit as an in process DLL that will be access though COM interfaces.

The License Manager communicates with four other logical units in the eStream Client. The interface with the Client UI control panel is through the ILicenseSubscritpionManager. This interface provides a complete list of all ASP accounts, subscribed applications, and accounting information to the Client UI control panel.

The interface with the App Install manager provides lists of application files when a new application is subscribed. These lists are stored in a database table. When an application is started access tokens are requested for the files that are part of the subscribed application.

The interface with the client network provides a connection to the eStream server that will supply the application file binaries to the eStream Client. The function of the LSM is to request lists of access tokens.

Threading Model

In order to service token requests and present application subscription information to the Client UI in a timely manner the License Subscription Manager will need to make use of multi-threading. Currently three threads are planed to fulfill the design requirement of this component. The main thread will satisfy command requests from the Client UI and Cache Manager, and App Install Manager. A separate thread will be spawned when the License Subscription Manager starts to handle Access Token Renewal. A new thread will start for every access token requested or renewed by the Cache Manager.

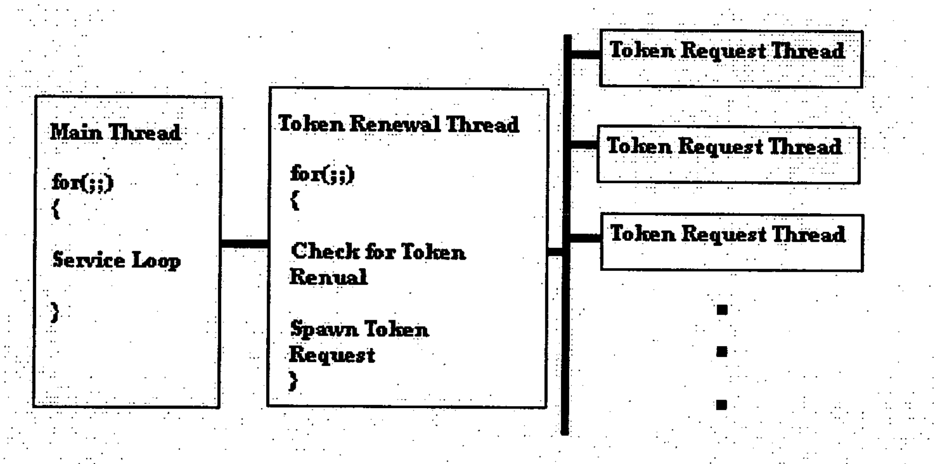


Figure 2 LSM Threading Design

The threads that provide License Subscription services will use Win32 SDK Event semaphores to signal to each other event notifications such a token renewal, network timeouts and token denial.

Main Thread

The main thread provides the interface support for the ILicenseSubscritpionManager and ITokenManager interfaces. When the main thread begins a worker thread is started that clears the token table by releasing and tokens that remain from the last instance of the License Subscription Manager. The token renewal thread sleeps on a timer waiting for an Access token to reach expiration.

```
/*
This is a psuedo code example of how the LSM main Loop will look like.  The
complexity of a COM implementation of this program unit means that the real code will
look very different from this code.
*/

LSMMainThread()
{
        Clear_Token_List();
        Start_Token_Renewal_Thread();
        For(;;)
        {
                Wait_For_Command();
                Switch(CommandType)
                {
                        Case SubscriptionManagerRequest:
                                Service_Request();
                        Break;
                        Case AccessTokenRequest:
                                Get_Access_Token();
                        Break;
                        Case AccessTokenGranted:
                                FireAccessTokenGranted();
                        Break;
                        Case AccessTokenDenied:
                                MessageBox(No Access Token);
                                FireAccessTokenDenied();
                        Break;
                        Case AccessTokenExpired:
                                MessageBox(Expired Access Token);
                                FireAccessTokenExpired();
                        Break;
                        Case Shutdown:
                                Release_Access_Tokens();
                                Terminate_Token_Renwal_Thread();
                                Return;


                }
        }
}
```

Token denial Policy and token expiration policy are two of the most critical issues that the License Subscription manager must handle. The policy for a token denial is to prevent

the user from running the subscribed application. The policy for token expiration is more difficult. Currently the plan is to nag the user into renewing their expired subscription using message boxes. We may move to some other policy as the License Manager develops.

Token Renewal Thread

The token renewal thread is responsible for maintaining the current list of tokens and requesting renewal for each token as it expires. Each time a token expires a new Token Request Thread is started to access the Cline Network Interface for a new Access token from the eStream Server.

```
TokenRequestThread()
{
        InitialIizeTokenTable();

        For(;;)
        {
                WaitForMultipleEvents();
                Switch(Event)
                {
                        case TimerPop:
                                CheckForExpiredToken();
                                For(each expired token)
                                {
                                        SpawnTokenRequestThread();
                                }

                        Break;
                        case TokenRequest:
                                SpawnTokenRequestThread();
                        Break;

                        case TokenGranted:
                                SetTokenGrantedEvent();
                        Break;

                        case TokenExpired:
                                SetTokenExpiredEvent();
                        Break;

                        case TokenRefused;
                                SetTokenRefusedEvent();
                        Break;

                        Case Shutdown:
```

```
                    Kill_Network_Threads();
                    Release_Tokens();
                    Clean_Up_TokenTable();
                    EndThread();
                Break;


            }
        }
}
```

Token Request Thread(s)

A token request is spawned by the Token Renewal thread and it runs until one of the following conditions is met.

1. The network client performs a timeout.
2. The Access Token is granted
3. The Access Token is refused.

A class pointer passed to the thread from the LPARAM function argument provides the actual token that the thread is requesting.

```
UINT TokenRequestThreadProc( LPVOID pParam )
{
    CRequestToken* pToken = (CRequestToken *)pParam;

    if (pObject == NULL ||
        !pObject->IsKindOf(RUNTIME_CLASS(CMyObject)))
    return 1;   // if pObject is not valid

    // Establish a connection with the Client Network Interfaces.

        IServerTokenManager.CreateDispatch()
        IserverTokenManager.GetToken()

        If (Timeout)
                SignalTimeOut();
        Else if (Granted)
                SignalGranted();
        Else if (Refused)
                SignalRefused();



    return 0;   // thread completed successfully
}
```

eStream App Server Low Level Design

Version 1.2

Sameer Panwar
[redacted]

Functionality

First, some definitions:

eStream page: the smallest unit of data that can be requested by a client from an App Server. Proposed to be 4kB for eStream 1.0.

page set: simply, a sorted list of eStream pages, each identified by a File ID (i.e. AppID & File #) and page # (essentially an offset into the file). This set is restricted only in that all pages in the set must have the same AppID.

client request: a single self-contained message from a client requesting a page set from the server. Each server response to a client request can return a number of pages, and there is a maximum number of pages that the client can request in this message. (TBD, somewhere between 8 and 20 or so).

The primary job of the App Server is to service client requests for application data blocks. The App Server is designed to minimize the amount of CPU time it must consume to satisfy each client request, thereby maximizing scalability. Thus, authentication is performed by a simple expiration time check of an AccessToken provided by the client, and compressed application data is saved persistently.

The App Server serves data derived from eStream Sets. To decouple the performance needs of the App Server from the Builder, we should have a post-processing tool that converts the flat, uncompressed eStream Sets as provided by the Builder into a precompressed format suitable for memory mapping, if the App Server is configured to serve compressed bits. Also, a profiling part of the App Server can be used to monitor for common page sets, and then assemble more optimized replies, which compress the set of pages together as a unit, to take advantage of improved compression ratios. These replies can be stored on disk to save time in rebuilding them each time the server is started up.

The App Server (AS henceforth) views an eStream Set as simply a set of files, and knows no further underlying structure. Thus an eStream Set contains at the start a table (FOST) indexed by File #, and providing the offset into the eStream Set where the associated file data begins, and the size of the file. So the AS just takes the client request of (AppID, File #, Page #, no. of pages), maps AppID to an eStream Set and looks up in the FOST table (File/Offset/Size Table) to find the requested data.

This works slightly differently when the eStream Set file has been pre-compressed by the post-processing tool. The resulting image is the same as before, except now the FOST points to another table, the POST (Page/Offset/Size Table). Because the compressed pages will be of different sizes, this table must by indexed by the Page # to find the relative offset and size of the compressed page data for the file. Thus if an AS is not configured for data compression, the main difference in behavior is that it doesn't do a POST lookup and it doesn't care about coalescing page sequences.

Data type & Data structure definitions

Processed eStream set – this structure is kept on disk and never changes after installation. It looks like:

```
struct {
     ApplicationID appID;     /* for reference, is a 128-bit GUID, see ECM LLD */
     uint32 maxFileNo;
     boolean compressed_flag;   /* indicates whether the AppFiles are compressed, though maybe we should do it differently? */
     FOST_Entry FOST[<maxFileNo>];
     uint8 appData[<sum of all AppFile sizes, which are variable>];
} ProcessedEstreamSet;
```

Since the files in the application are of variable size, we can't make a table out of them, and must indirect out of a table (indexed by the File #) to find their offset location inside the AppData buffer.

```
struct {
     uint32 offset;
     uint32 size;
} FOST_Entry;
```

When the processed eStream set is compressed, then we use the AppFileCompressed structure at the offset indicated by the FOST, otherwise we interpret the data as just AppFile. The AppFileCompressed structure starts with a table that indicates the size and offset of the compressed data that belong to the page it was indexed by.

```
struct {
     uint8 fileData[<size from FOST_entry>]
} AppFile;

struct {
     POST_Entry POST[<number of pages, derived from size from FOST_Entry>];
     uint8 fileData[<sum of all FilePage sizes, which are variable>];
} AppFileCompressed;

struct {
     uint32 offset;
```

```
    uint32 size;
} POST_Entry;
```

This covers all the structures that live on disk. When we mmap-per-file, that means we make multiple mappings out of a single ProcessedEstreamSet file, at different offsets, one for each file.

Now, for the in-memory data structures (assuming per-file-mmapping):

The primary lookup will be a hash table, hashed on the AppID and FileNo. It should have on the order of 10,000 entries, each table entry containing a list of entries (for collisions). Each list entry contains:

```
struct {
    ApplicationID appID;
    uint32 fileNo;
    uint32 size;   /* size of the mapped Appfile */
    MMap fileMap;
    HTListEntry * next;
} HTListEntry;
```

The Mmap struct just contains any OS-specific-related stuff to manage the mappings, plus a field `char * ptr`, which points to the place in memory that the AppFile (or AppFileCompressed) is mapped. So the hash table looks like:

```
struct {
    HTListEntry * entry[<size of hash table>];
} MMapHT;
```

Hash function is TBD. The hash table should be statically sized large enough to handle the full number of eStream sets up to the maximum memory we will support. Assuming 32 bytes being used per entry, that implies about 1 MB to handle 30k files, which is no problem. (Maybe we should reserve entries for 100k files or more?)

Configuration: Each AS must obtain configuration data, either directly from the database or from the monitor in its startup message. The required data is (with the config param names and datatypes):

```
AppList         vector of ApplicationID's (128-bit GUIDs)
ServerPort      uint16
MonitorPort     uint16
SLiMKey         uint (size TBD, depends on actual algorithm)
ClientTimeOut   uint32
CompressionFlag uint32
```

Network communication: The AS talks only to clients and the server monitor via the network. The server monitor communication will be described as part of the monitor heartbeat protocol. The AS-client communication will be described in a separate docu-

ment. The AS will time-out and close connections that have been idle for some amount of time (a few seconds).

[maybe combine multiple responses into a single send socket call (will only work for TCP probably, since proxies won't like multiple server responses)?]

Interface definitions

The AS is optimized to do one thing only: serve pages from the read-only file system part of eStream, so there is just one interface with the client. Anything the client can care about in an eStream set is just another file to the AS, including the AppInstallBlock, and directories/metadata. The AS only returns the data the client requested, nothing extra.

```
struct {
     uint32 fileNo;
     uint32 pageNo;
} PageRequest;

struct {
     uint32 errorCode;
     uint32 compressedFlag;
     uint32 fileNo;
     uint32 pageNo;
     uint32 offset;  /* offset into pageData below */
     uint32 dataSize;
} PageReply;
```

PageReadRequest

Caller:	Client	
Callee:	AppServer	
Input:	uint32	appId;
	eStreamAccessToken	accessToken;
	uint32	numPagesRequested;
	PageRequest	pageSet[(numPagesRequested)];
Output:	uint32	numPagesRequested;
	PageReply	pageSetReply[(numPagesRequested)];
	uint8	pageData[(sum of all page data)];
	uint32	globalErrorCode;
Global Errors:	INVALID_ACCESS_TOKEN	
	EXPIRED_ACCESS_TOKEN	

INVALID_APP_ID
EXCEEDED_MAX_REQUESTABLE_PAGES

Errors within
PageReply: INVALID_FILE_NO
INVALID_PAGE_NO
SERVER_ERROR (probably should be logged, and should cause an alert if too many occur in some time period, including errors that don't get returned to the client.)

AppServers don't ever talk to the database (it would be a waste of licenses considering the number of AppServers we'd have and their infrequent accesses). Instead, they obtain all their relevant control information from the server monitor.

The exact interfaces are TBD, but from the monitor they will provide configuration information, AppServer state change requests, and add/remove requests to the list of apps being served. Going back from the AppServer to the monitor, it will report load (average response time) on a per app basis, and server state, along with the heartbeat.

Component design

Interesting issues to deal with:

Scalability/Performance

Since scalability (and thus performance) is critical for the AS, let's go over how CPU and memory are used.

Memory

Performance is maximized when virtually all client requests can be satisfied by retrieving the desired pages from RAM, because RAM is far faster than disk. Thus the amount of RAM available will put an upper bound on the number of apps that a single AS can serve efficiently. Since server RAM won't grow as fast as the total size of all apps available as eStream sets, this means we'll have to heterogenize servers, where each server specializes in a subset of apps, limited by available RAM. For eStream 1.0, this component of AS configuration will be handled manually, the eStream administrator assigning apps to servers. In the future, the set of App Servers should automatically reassign apps dynamically to balance load.

But this is just one level of memory, committing RAM to a set of apps. There still remains the question of how to best utilize that RAM for each app, since some files are used far more often than others. This immediately means that for efficiency we must overcommit RAM, because if we allocate an entire eStream set into RAM, we're using precious resource to hold data that may be requested only very rarely. Instead of having to manage our physical RAM manually to accomplish this (such as with a cache), an easier approach would be to take advantage of virtual memory (VM) to automatically keep

the hot pages in RAM, with the remainder available (again **automatically**) off disk (via memory mapping the eStream sets). That way the server can satisfy any possible client request for any app it serves, but is optimized to be the most efficient over all clients. But this only works if enough VM is available. (Time for some back-of-the-envelope numbers.) Given that an app seems to have something like only 20% of it being hot (from our current limited data from the prototype), this means VM must be at least 5x of RAM for maximum efficiency. Given that a process has about 2 GB addressable VM, this corresponds to about 400 MB of RAM. Beyond that size (which is not uncommon), we don't have enough VM to efficiently overcommit our memory (by mmaping entire eStream sets). So now our choice is to either manually manage a memory cache (and all the attendant coding, bugs, etc.), or to mmap at a finer granularity.

Note that the effective virtual memory required by an app is increased when compression is used, to handle the extra compressed page sets. They'll probably double or triple the RAM footprint by hot pages (due to redundancy), but only increase the overall VM footprint by 1.2 – 1.5. The consequence of this is that the overcommit ratio goes down to 1.5 / (3 * .2) = 2.5, though the amount of apps servable is reduced to 1/3 (!!). Now 2 GB virtual address space corresponds to 800 MB of RAM. This means we should be able to just memory map entire eStream sets, up to 2 GB worth, and be confident we're utilizing RAM efficiently, assuming the server has about 800 MB of RAM. A server with less RAM will likely thrash, and those with more will likely see little improvement in the number of apps they can serve via memory mapping.

A loss of 2/3 in the number of apps an AS can serve I think is too great a sacrifice, too great a loss in app scalability (need 3x the number of servers as before!) for what is about a 15-30% greater effective bandwidth at the client. The root of this problem is the redundancy (costly in physical memory), because the compressed page sets will contain the same page in multiple sets. This is similar to the redundancy that appears in trace processors and dynamic translation, which places extra memory demands in both those cases. I think we must completely eliminate this redundancy to achieve the goals we desire, either by (1) not using compressed page sets, and just sending multiple individually compressed pages, or (2) ensuring a page appears in only one compressed page set. [There further potential loss of effective memory size when using compressed page sets since they'll be allocated in 4k chunks, thus wasting about 2k on average; we'd have to batch them up together in files to minimize this… Also, saving the compressed page sets to disk introduces extra complexity to the AS because we'd have to properly handle recovery (i.e. what if the system crashes while we're writing the sets, which if we're memory mapping is totally out of our control). Because of this robustness requirement, and the fact we need to be 100% sure we're serving good bits (lest we crash a bunch of clients), this needs to be thought out very carefully if we want to do this. My opinion is that we should defer implementing compressed page sets until we better understand the tradeoffs, and good profiling schemes. In particular, will the AS be mostly bandwidth-limited, memory-limited or CPU-limited?]

Separately from this, we should consider the effect of per-file memory mapping (ignore the compressed page sets now). This has the impact of requiring many more mmap's

from the OS, but promises better use of the limited virtual address space. In this scheme, we mmap each file into VM as it is referenced by a client. If only hot files are referenced, then the RAM footprint is the same as before, but VM is only used for the hot files, not the entire app, probably about 30-50% greater in size. Thus the overcommit ratio then becomes 1.5, much better than the 5 with full app mmapping. So 2 GB of VM corresponds to 1.3 GB of RAM, much better than the 400 MB with full app mmapping. However, this assumes that VM is used in a cache-like manner, evicting not recently used mmap's, since as uncommon files are referenced, they eat up more and more VM. Once VM is totally used up, then replacement policies and eviction (and fragmentation of virtual address space) become issues, just as with a manually managed cache. One solution is to simply purge all mmap's and start from scratch, which is simple and reliable, especially considering the AS is multithreaded (if this is done, the above analysis doesn't hold, and performance becomes a function of how often VM is cleared). Another possibility might be to use the profiling mechanism and only place sufficiently popular files in mmaps and do regular file system accesses for the rest.

Of course, the alternate option for managing physical memory is to know its size, and manage a cache manually. One advantage here is that the AS would know the physical memory consumption and usage (unlike when the OS was handling everything), which may help with load balancing. The main advantage is that there are no artificial limits (overcommit ratio is irrelevant), and only physical memory size is the true limit, and this approach can map any number of eStream sets (with any size of files) to any amount of physical memory up to the virtual address size (4 GB). Then memory management becomes an issue (what do you do once all your RAM is full), which can be painful in a multithreaded environment. Again, we can just invalidate the whole cache as an option, but this will probably happen more often than with the per-file-mmapping case, unless RAM is greater than the maximum that the per-file-mmaping approach can handle. If the wholesale cleanup approach is used, then allocating fixed size chunks may not be needed, and we could potentially get better memory usage by packing compressed pages more tightly (e.g. 16-byte aligned vs. 4kB aligned), which is another potential advantage. Maybe instead of wholesale cleanup, we mark the most commonly used pages, and then just compact those and dump the rest (say 50%). The main issue with this approach is potential redundancy with respect to the OS disk cache (which is shared in the mmap approach), and assumption that our caching policies will be better than the OS's. Also, lookups get messier, since we need a bigger lookup table to index via page # as well.

Yet another option is to use multiple processes instead of multiple threads, one process per app being served, thereby releasing us from the 2 GB VM limitation. However, this introduces the issue of multiplexing requests from the network via IPC, and more load on the server monitor. On x86 NT, a Very Large Memory feature is available that can provide 36-bit addressing per process; we may want to use this even though it won't be available on regular Unixes (and probably not x86-linux).

In summary: per-eStream-set-mmapping is probably too wasteful of virtual address space. Per-file-mmapping is much better, but then memory management becomes an issue, suggesting a simple throw-away-and-start-over solution. However, given that solu-

tion, if a lot of physical memory is desired, a manual cache approach may be better (the better packing should overcome any loss due to redundancy with the OS disk cache). Compression of page sets invokes several issues that probably can't be fully addressed until after 1.0. **Bottom line: the target now for 1.0 is to use per-file-mmapping with per-page compression (but no compression of page sets).** Also, we should instrument the system to allow us to easily collect the relevant data (mem usage, CPU load of different routines, etc.) to help guide us in further evolution of the system to improve performance (e.g. compressed page sets or explicit page cache).

CPU

The main work of the CPU is as follows (encryption is assumed to be done by hardware since its CPU impact is severe):

1. OS system call to retrieve request from network.
2. Decode client request.
3. Validate AccessToken.
4. Lookup AppID, File # in primary lookup hash table. (If mmapping eStream sets, instead lookup in App table, then lookup in FOST).
5. If mmapping then (if uncompressed, no further lookup, if compressed, then lookup in POST to find page and size), if explicit cache then look in B-tree (secondary lookup).
6. If lookup fails, then bring in the data off disk (either mmap or file system call).
7. Copy page data to reply buffer.
8. OS system call to send reply to network.

However, if compressed page sets are used, lookups get more complicated, with a different set of tables to check for an appropriate page set first (and lookup failures incur potential decompression/compression). It appears the least amount of CPU time is probably incurred when doing per-file-mmapping. All pages held in memory are kept in compressed form to save repeated compression of the same data, so pretty much all the work is in lookups and memory copies. Potentially the AccessToken validation will use hardware assist. Lookup failures (i.e. having to go to disk) should be relatively uncommon, and memory should be sized to ensure that.

However, since the AS will run in user mode, this incurs the penalty of two extra copies (from the network buffers) and switching between kernel and user mode twice. If this is enough of a problem, we'll have to consider implementing the AS to run in the kernel (all commercial NFS, etc. implementations run in the kernel), which means we should choose our implementation to be compatible with that approach. In particular, we may not be able to rely on the virtual address space not being fragmented, so mmapping full eStream sets may be impossible. Plus robustness of the server becomes even more important, and portability issues arise. For the 1.0 release, we plan to implement the AS in user mode keeping the possibility of moving to kernel mode in the future, and will collect data from 1.0 (or derived prototype) to evaluate the actual benefits.

Disk: Since we are relying heavily on the common pages being in memory, we could possibly even consider storing the processed sets on a network disk, i.e. remote from the app server itself. However, such sharing won't work well for compressed page sets since

they are written to at runtime—it would be extremely messy to handle dozens of app servers trying to add many compressed page sets (possibly the same) to a set of shared files.

Multithreading model

The approach will be to have a single boss thread which pulls things out of the network port and stuffs client requests into a queue and a bunch of worker threads which grab requests and send back the replies. Simple enough, but this raises the issue of thread control, since the boss also needs to be able to handle threads that die or hang and kill and restart them. The boss thread will monitor the worker threads and provide load/hearbeat info to the monitor through the server manager thread, thus giving visibility to the server monitor of the health of all the worker threads.

Load balancing

To be described elsewhere? (appears in SLiM server LLD)

Security

There are two levels of security involved in the AS. First, we must prevent clients who don't hold valid licenses from gaining access to the licensed binaries. This is accomplished by the client obtaining an AccessToken from the SLiM server and presenting it to the AS upon every request. The AS can then use the SLiM server's public key to test the authenticity of the AccessToken (to protect against forgeries), and then can test the authentic expiration time of the AccessToken. Second, we must encrypt the actual data being sent on the wire to prevent third parties from gathering the binary data covered by the license. Since the data coming out is somewhat obfuscated anyway (files are identified by arbitrary IDs, with our own strange message formats and compression and all in random pieces, etc.) it is not clear how much extra protection is really necessary, i.e. what do the license issuers actually want? We should use a common scheme like SSL to perform this encryption. It has been decided that the encryption load for this would be too great, and thus the data send back will be unencrypted. We may use SSL for authentication purposes only (i.e. null-cipher), if that is cheap enough.

Also, a possible optimization for checking AccessTokens would be to cache recently used AccessTokens along with a signature/hash. If a token presented by a client matches, then we can skip the authentication step (since we've done it once already) and just check the expiration time.

Robustness

The AS must be very robust. It must catch OS call errors and handle/log them as appropriate, and deal with threads that hang or die. Thus it needs to aggressively check for error conditions and possible failure modes. The AS also needs to track relevant resources (e.g. sockets, memory) and carefully manage/reclaim them so as not to exceed any limits or to degrade performance. And of course, the AS needs to check all data coming in from the client, to deal with ill-formed requests, and illegal values (e.g. huge negative indexes, etc.), and perform no potentially dangerous operation without validating parameters. This becomes even more important when we eventually move the AS to run in kernel mode. The AS also needs to be as stateless as possible, to minimize recovery time, and if it does perform writes to disk (such as for the compressed page sets), do so in a reliable fashion conducive to quick recovery. Any unreliability in the AppServer will negate any benefit of scalability we have over our competitors.

Testing design

This document must have a discussion of how the component is to be tested. Some subsections could include:

Unit testing plans

The various components of the AS are not too large or complicated: The request dispatcher (to worker threads), the hash table, the compression code, the AccessToken checking code, etc. These shouldn't be too hard to do reasonable testing on in isolation.

For the post-processor component, we'll have to build some sample Estream Sets as input, but it'll be hard to tell whether the output is correct without having a minimal working AS.

Cross-component testing plans

The best approach will be to perform incremental implementation and testing. I.e. we build the core functionality that is required (i.e. can start with just regular i/o reads), and then add the more performance-related stuff later (adding mmaping, and then the hash table & AccessToken checks), while testing the entire system as pieces are gradually added (of course performing sanity-check and other minimal testing on the pieces first if possible). Compression can be added last.

To actually drive the AS, we'll need a test client, which will be designed to just shoot off a series of read requests to the server. The file data returned could then be written to files, and this can be compared against the original set of files used to create the Estream Set we started with, to check that the data was received properly. For checking error conditions, a log of errors can be written and compared against a reference log for those requests we expect to fail.

Stress testing plans

To accomplish this, we should run multiple independent test clients (on the same machine and on different machines), and increase the frequency of requests (to stress the AS's threads and sychronization, and communication routines), and the number and size of files referenced (to stress the hash table and memory). Each test client can then check whether the data and errors it got back were as expected, like in the above subsection.

Coverage testing plans

Should we use some kind of code coverage tool for this?

Performance testing plans

Since performance is critical, we should take the time to evaluate the AS's performance characteristics. We need to crank up our stress testing until either bandwidth or CPU saturates, and record the request rate that generated it. We should compare how this point responds to high numbers of clients with fewer requests per client vs. fewer clients with higher requests per client. We'll need to profile the system to find bottlenecks to tweak more performance out of it, and learn how well our original design assumptions hold up. Depending on whether CPU or bandwidth (or memory) saturates first, we may want to modify the system's tradeoffs to improve scalability further, and otherwise note which components a customer should upgrade for better performance. Also, if we think we can come up with reasonable client access pattern profiles, we may want to use those to estimate the actual number of real-world clients an AS can support. As part of this, we'll probably want to run the AS in-house once it is mature enough (eat our own dogfood), and then farm out app upgrades, etc. (play out some of our scenarios) and see what happens to the AS's (do they choke or what).

Availability testing plans

We will also need to test our failover and load balancing capabilities. This will require several test machines with the monitor in place to start and stop servers, and have clients be aware of multiple AS's and respond appropriately when an AS stops responding. For load balancing, we'll probably want a bunch of test clients with a variety of access patterns and see how well their requests are distributed.

Upgrading/Supportability/Deployment design

App Servers will possibly need to version their interface with clients (requiring clients to state the version they're expecting), but will also need to support older versions. We may also modify the Estream Set format (or just the processed set format), but that should be handled by upgrading both the AS and post processor and then regenerating the processed sets.

For supportability & deployment, the AS will report error conditions and load to the server monitor, which is used by the customer.

Open Issues

1. Is there a limit to the # of possible mmaps?
2. Is there a single system call to unmap all mmaps?

eStream 1.0 CORBA Centric Server Framework

Authors: Amit Patel, Bhaven Avalani, Michael Beckman

Date[redacted]

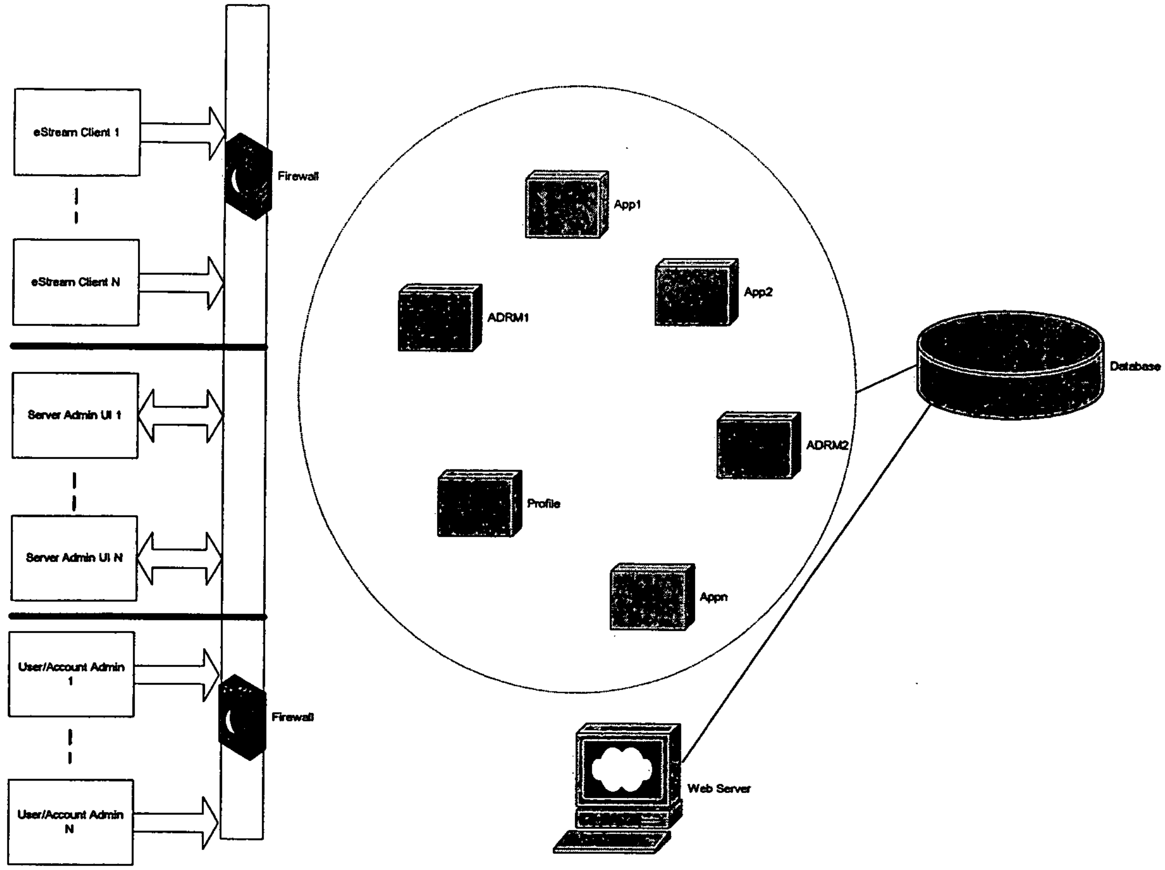
Omnishift Confidential

Abstract: The following document presents a server framework based on CORBA for eStream 1.0.

Descriptions: eStream 1.0 is a distributed server environment. CORBA provides a cross-platform cross-language distributed system solution. The high level essential features of the CORBA framework are listed below.

- Messaging Support. How do the servers and clients, servers and servers talk with each other. CORBA provides mechanism for inter-object communication on a variety of protocols (IIOP, GIOP, IIOP over HTTP).
- Distributed Object Management. This essentially is useful for management and monitoring in eStream as the client side objects eStream supports are fairly simple. However for management and monitoring all servers need to provide objects which advertise the health of the system.
- Services: Corba provides a variety of services for a distributed system.
    - Naming Service. Helps maintain the location of objects in the systems. This is very useful for server management tools.
    - Event Service. Useful for Alarms etc.
    - ORB service. Used for server configuration, server state. It has the capability to stop/start servers.
    - Security service. Useful to access control and encryption services.
    - Distributed Transaction Support. Probably not relevant to our framework.

The following diagram illustrates the eStream architecture at a very coarse level.

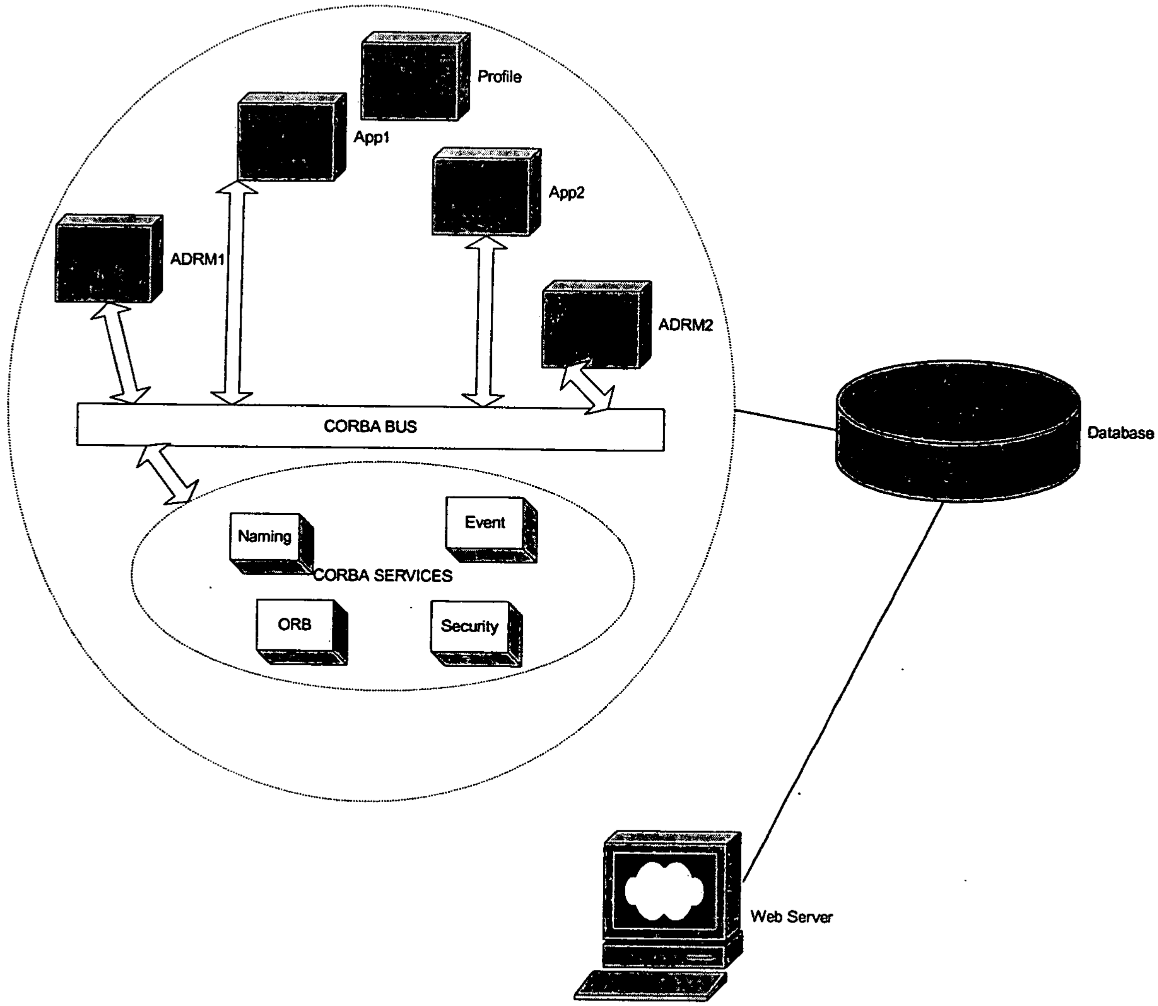


Listed below are the objects in our system and the data they manipulate.

CLIENT	DATA	LOCATION
eStream Client	Account/User/Subscription Information	RDB/LDAP
	EStream Sets	File System/RDB
	Server Information(Location of ADRM, APP, Profile etc)	???
User/Account Management Client	Account/User/Subscription Information	RDB/LDAP
	Server Information(Location of ADRM, APP, Profile etc)	???
Server Administrator Client	Server Information(Location of ADRM, APP, Profile etc)	???
	Real time/Heart Beat	???

	status of the servers	
	Load information for the servers	???
	Configuration information for servers	???
Servers (as Clients)	Static Configuration of other servers in the system. Give me a server which serves Word.	???
	Dynamic Configuration of other servers in the system. Heartbeat and the load are examples of this information.	???
	Load/Logging/Alarm Information. Log this access. Write down my load. Raise this alarm to the administrator.	???

The question marks in the table above are transient data which characterizes the current state of the servers in the system. A CORBA based system will solve this problem using the following server architecture.



The management client in this scenario will essentially talk to the CORBA system to get any information of the servers in our system.

Listed below are the pros and the cons for a CORBA based system.

PROS:
1. A well-defined and proven server framework.
2. Cross platform support.
3. Lot of services is available for free. Alarm, Management, Load Balancing.
4. Distributed. The objects in the system are inherently distributed and hence more scalable.
5. High performance system. Transient data about the system is stored in transient storage and hence data accesses are fast.

6. Tools and services are available for free. Example: A distributed transaction support system is available and may be useful for eStream in the future.
7. Server management and Alarm tools are easily available.

CONS:

1. Vendor Lock in. Visigenix and Iona are primary vendors with their own set of quirks. Both do not have a good history of migration support. (Partly due to the CORBA standard evolving very rapidly).
2. In house expertise.
3. The cost of the solution may be too high. (Need to investigate on this).
4. May a very complicated solution for a simple problem.

eStream 1.0 Database Centric Server Framework

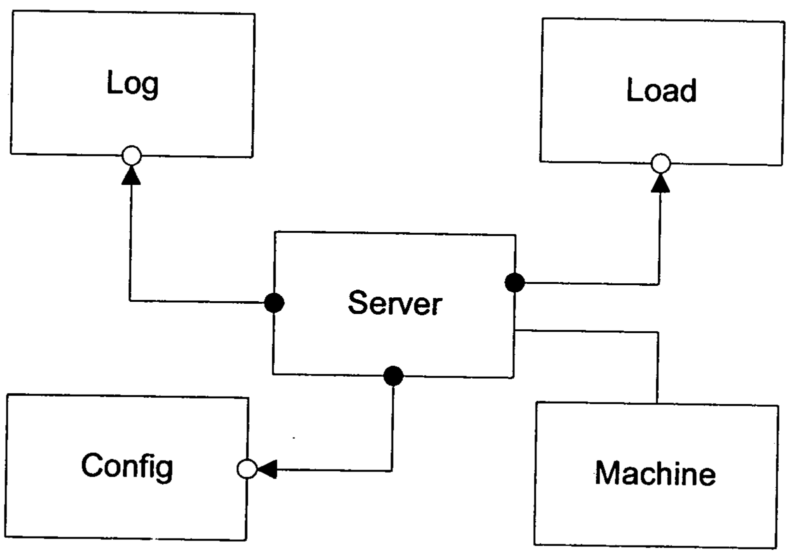
Authors: Amit Patel, Bhaven Avalani, Michael Beckman

Date: [redacted]

Omnishift Confidential

The data centric Server Framework Architecture uses the database as the center of all communication channels. The following diagram illustrates the architecture of and eStream system based on the database framework.

The following data model diagram illustrates the data model of a system management scenario in a database centric server framework.



Server: A logical server instance in the system. This will have the state attribute associated with it. The state attribute will describe the current state of the machine.
Config: A set of configuration values for a server. The configuration values can be hierarchical.
Machine: A physical machine. This will have a set of parameters describing the physical machine. IP, Physical memory etc.
Log: This will maintain the log for eStream servers.
Load: Used to record historical and real-time load on a logical server instance.

The interactions with the database in this model are denoted below.

1. Read/Write Configuration. This is used to update the configuration table for a given server instance. Reading the configuration data will involve a simple select from the configuration table.
2. Read/Write Machine Records. Create and update physical machine information. Not done frequently.
3. Read/Write Log information. This is done once per heartbeat. A batching mechanism may be appropriate here. This is done once per heartbeat. A batching mechanism may be appropriate here.
4. Read/Write Load information. This is done once per heartbeat. A batching mechanism may be appropriate here. This is done once per heartbeat. A batching mechanism may be appropriate here.
5. Read/Write Server information. Done when creating new server configurations. It may be useful to duplicate some information here for faster access. (eg The most recent load characteristics.).

<u>Thoughts on installation of server components</u>

- All server side components including server side client interfaces with the exception of the database to be installed on every machine within a deployment for estream v1.0 (for future releases we can consider something more elaborate)
- monitoring process will be started (the server client admin can launch this from its UI), find and connect to database, configure itself, and then configure and launch logical servers.
- server side clients can be launched independently of the monitoring process.
  - Server side clients can configure and read/write from the db even if the monitoring process and all logical servers are not running.

<u>Thoughts on Monitoring process</u>

- A monitoring process is required to monitor the state of the logical servers.
- monitoring process is responsible for load balancing and maintaining high levels of QoS.
- One monitoring process per deployment installation.
  - monitoring process can monitor across machines
- monitoring process can re-launch a server side process (this can be a configurable option)
- monitoring process can be used to start and stop server side processes
- Monitor requests heart beat request for each logical server. If failure to respond in reasonable time, it will post an alarm in the database.
- heart beat request rate is configurable.

- heart beat request has a very simple and easy to decode protocol which may include:
    - simple request pulse
    - stop request pulse
    - dynamic config request pulse
- Each pulse reply may include current load information from logical servers.
    - key servers to monitor load is DRM and App
- Monitor needs to track load of DRM and APP minimally.

Random thoughts on db updates and logging

- log atypical events instead of typical when logging process state changes.
- DRM (all server processes in general) to maintain performance critical information in memory as well as in db for quick response. This ensures persistent state if the DRM crashes yet hopefully allows for very fast response.
- Assume that the performance requirements on the webserver are such that it can read/write to the db on every transaction if necessary.
- Access token to maintain the app server for getting requests. Client needs to do quick check on app server to see if there is change. If so client needs to re-direct to different app server.

Thoughts on configuration

- Server configuration can only be performed by a server-side administrative client.
- Typically, all configurations go through the db where they are persistently maintained.
- administrative client changes causes table updates in the database.
    - client could send direct socket update to monitor process (if it is running; can be determined by client from db) for requests for dynamic configuration change.
- monitoring process reads tables in the database for configuration information and will launch server side processes as appropriate.
- Once a logical server is up an running, modifications to the configuration are either:
    - read directly from the db by the logical server and acted on (polling)
    - monitor sends special heart beat request which states that the logical server should recheck its configuration. (event driven)
    - monitor shuts down process and restarts to pick up config change through the normal start-up process
    - monitor communicates configuration through heart beat protocol.

Issues to Resolve

- How would the monitoring process remotely start a server process.
    - once process is started monitor knows since it will respond to the heart beat request that will be sent after process is launched.
- How does monitor get launched. Monitor needs to get to the database and access with proper password.
    - more detailed config information for monitoring tool can be gathered from database.
- Need to come up with some concrete examples showing dynamic config update.
- Need to make distribution system proposal
- Ask Anne to determine what the data sets for profiles will look like.
    - will these profiles go into the data base or flat files.

eStream Configuration Management Low Level Design

Bhaven Avalani
[redacted]

Functionality

The configuration management utility is used by all server components to manage the server configurations. It provides the following functionality:

- Configuration for a server consists of a set of name – value tuples where the values themselves can be a set of name-value tuple.
- Servers can load the complete configuration from the database.
- Servers can load the configuration for a given name.
- Server should be able to load the configuration from a flat file also.

On the Server Manger interface, configuration will appear as a table containing name – value tuples. The table may be hierarchical to represent nested structures containing the values which can themselves be name values. An example of a simple name-value pair would be:
PORT 8080
An example of nested name values would be:
Applications:
    word.exe   windows2000sp3
    excel.exe   win98sp4
On a flat file the configurations will always be name-value pairs. To represent one level nested structure the format would be:
Applications word.exe   windows2000sp3
Applications excel.exe   win98sp4

Data type definitions

```
Class  tuple {
        string name;
        Value value;
};

class Value {
        int type;
};

class StringValue: public Value{
        string value;
};
```

```
class TupleValue: public Value {
        vector <tuple> tupleArray;
};

typedef vector < tuple > configArray;

class ServerConfig {
private:
        configArray Array;
public:
        ServerConfig(); // Default initializer
        ServerConfig(string filename); // To initialize from a file.
        ServerConfig(ServerId, Dsn, Dbuser, Dbpasswd); // Initialize from DB
        configArray* GetConfigArray(int serverId);
        tuple FindConfig(int serverId , string name);
        Reload(int serverId);
        GetConfig(int serverId ,string name);
};
```

Interface definitions

GetConfigArray

configArray* GetConfigArray(int serverId);

Inputs:

int serverId

Outputs:

An array containing the configuration information.

FindConfig

tuple FindConfig(int serverId ,string name);

Inputs:

int serverId

Name of the configuration to find.

Outputs:

The tuple containing the name and value for the config

Reload

void Reload(int serverId)

Comments:

Reload all the configuration for the server information from the database.

GetConfig

void GetConfig(int serverId , string name);
Comments:
Reloads the configuration for a given named configuration.

Component design

Testing design

Unit testing plans

The configuration module is a stand-alone module which will be tested using a config-test.exe executable. The executable will exercise all of the interfaces described above. The configtest executable should be testable in the DB and the non-DB mode.

Stress testing plans

Not relevant to this component.

Coverage testing plans

Cross-component testing plans

Upgrading/Supportability/Deployment design

Open Issues

eStream HTTP Protocol Low Level Design

Bhaven Avalani, SameerPanwar

[redacted]

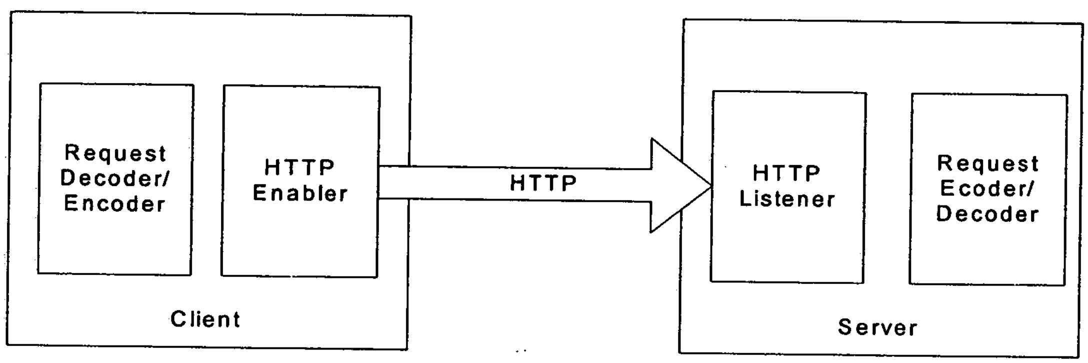
Functionality

The eStream client will talk to the eStream servers using the HTTP protocol. This allows the eStream clients to run across a firewall. Since, the clients and the servers are both developed by the OTI, we will implement the minimum HTTP protocol to optimally server our environment. The subset of the HTTP protocol we will implement is:

HEADER

- POST primitive. (GET/LOAD are not applicable to our situation as we will always post our data structures in pre-defined format.).
- Keep-Alive primitive. (Needed for maintaining the connections).
- Host primitive. (Needed for maintaining the connections).
- Content-length primitive. (Needed to access the POST data).

The process of client server communications over HTTP is explained in the diagram below.



The following example shows the lifecycle of a request in this model.

1. Client requests for a file. GetFile(string fileID, string version);
2. Encoder encodes the request into a bitmap structure(To be defined).
3. HTTP Enabler plops in the following header to the request:
   POST / HTTP1.1
   Host: <servername>
   Conneciton: KeepAlive
   Content-length: <content_length>

<content>

4. Listener gets the request and parses the request get the content length etc.
5. The content is forwarded to the Request Decoder.
6. Response is generated on the server.
7. Http Listener plops in the HTTP header to response.
8. Client gets the response back.

Based on the discussion above there are essentially three major components in this architecture:

1. Request Encoder/Decoder. Takes C type requests and encodes them to a binary format.
2. HTTP Client: Makes simple HTTP 1.1 requests.
3. HTTP Listener: Receives request, parses them and then forwards them to the decoder.

Data type definitions

Interface definitions

void sendMessage(char* mesg, char* ip, int port = 80, char reply)**

Inputs:

mesg: The message buffer to be sent.
ip: IP address of the server to send the message to.
port: Port number to send the message to. Default is port 80.

Outputs:

reply: The reply from the server.

Description: The HTTP client will make this request to send a message to the server.

Errors:

IO Exception. Occurs when the message is not deliverable.

void readRequest(HTTPRequest* Req)

Outputs:

HTTP RequestStructure.

Description:

Called on the server side to read and parse the incoming request on server listener socket.

Errors:

INVALID_REUQEST (We do not support GET and LOAD)
IO Exception. Socket error on receive.

void sendResponse(char* mesg, HTTPRequest* Req)

Inputs:
mesg: The response message to be sent.
Req: The Request structure containing the original request.
Description:
Called by the server to send a response to a request.
Errors:
IO Exception. Socket error on send.

Component design

Testing design

This document must have a discussion of how the component is to be tested. Some subsections could include:

Unit testing plans

Stress testing plans

Coverage testing plans

Cross-component testing plans

Upgrading/Supportability/Deployment design

This document must have a discussion of how the component addresses any specific issues related to upgrading, supporting and deployment of e-stream applications. Some examples include: error conditions detected and reported by this component, any special hooks this component will provide for monitoring, hints for troubleshooting problems, any special hooks for debugging this component.

Open Issues

This is a list of issues that need to be further investigated or revisited during implementation.

eStream Logging Low Level Design

Bhaven Avalani

[redacted]

Functionality

All servers and clients in eStream 1.0 need to log the error and access data. Logging enables component debugging and auditing support.

EStream Framework should provide logging with the following features:

- Each component will have an error and optioanally an access log file. The names of these files would be <component>_error.log and <component>_access.log.
- The files will be located in the <eStream1.0 Root Dir>\logs directory.
- The error log files will have messages with the following priorities:
    - 1-Low : A warning which can be ignored.
    - 2-Medium: A warning which needs to be looked into.
    - 3-High: Recoverable error in the component.
    - 4-Critical: Irrecoverable error. Needs admin assistance.
- Logging level should be configurable. The following levels are to be supported.
    - 0: Only errors will be logged. This will be the default level.
    - 1: Errors and Warnings to be logged.
    - 2: Errors, Warnings and Debugging information to be logged.
    - 3: Errors, Warnings and advanced Debugging (like memory dumps, tcp stack dumps etc) to be logged.
- Log Wrapping to be supported. The log files will wrap at a predefined size. On wrapping the following actions will occur:
    - Any existing <logfile>.bak will be deleted from the system.
    - The current <logfile> will be backed to <logfile>.bak
    - The component will continue logging to the <logfile>.

For each eStream client and server component logging the log files (component_error.log and component_access.log) should be written in eStream1.0Root\logs directory. The formats for the log files will be as follows:

Error Log:

[HEADER]

[Thread ID] [TimeStamp] [Priority] [Message]

[FOOTER]

An example of this log format would be:

```
*********************************************
Omnishift eStream Application Server
Server Started.
StartTime: 14/Aug/2000:16:31:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
*********************************************

0 [14/Aug/2000:16:31:19 -0700] 3-High Cannot connect to the database.
Invalid Username/Password.
1 [14/Aug/2000:16:31:19 -0700] 4-Critical Cannot start the HTTP listener
at port 80.
0 [14/Aug/2000:16:31:19 -0700] 4-Critical Shutting down the server.

*********************************************
Omnishift eStream Application Server
Server Stopped.
StopTime: 14/Aug/2000:16:35:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
*********************************************
```

Access Log:

[HEADER]

[Thread ID] [TimeStamp] [Message]

[FOOTER]

Data type definitions

typedef enum{1-Low, 2-Medium, 3-High, 4-Critical} ErrorLevel;

typedef enum{0,1,2,3} LogLevel;

Interface definitions

SetErrorLogFile: Set the error Logfile.

Bool **SetErrorLogFile**(string filename)

Input:

Filename: Name of the file.

Output:

Success/Failure

Error:

I/O FAILURE

SetAccessLogFile: Set the error Logfile.

Bool **SetAccessLogFile**(string filename)

Input:

Filename: Name of the file.

Output:

Success/Failure

Error:

I/O FAILURE

SetErrorLogFileSize. Set the error log file maximum size.

void **SetErrorLogFileSize(** int fsize = 10)

Input:

fsize Size of the file in MB. Default is 10.

Comments:

If this API is not invoked, then the file size defaults to 10 MB.

SetAccessLogFileSize. Ser the access log file maximum size.

void **SetAccessLogFileSize(** int fsize = 10)

Input:

fsize Size of the file in MB. Default is 10.

Comments:

If this API is not invoked, then the file size defaults to 10 MB.

SetErrorLogLevel. Set the log level of logging errors.

void **SetErrorLogLevel(** enum LogLevel = 0)

Input:

Loglevel enum defined above. Default is 0.

Comments:

Can be called any time during the execution to change log level.

LogError. Log an error message. This interface will take in variable arguments.

void LogError(long threadid, enum ErrorLevel, char* format, ...)

Input:

Thread id if the caller thread.
Error level. Enum defined above.
Format. Printf like format.

Variable number of arguments.

Errors:

INTERNAL IO ERROR

LogMessage. Log an access message.

void LogMessage(long threadid, char* format, ...)

Input:

Thread id if the caller thread.
Format. Printf like format.
Variable number of arguments.

Errors:

INTERNAL IO ERROR

Component design

Testing design

This document must have a discussion of how the component is to be tested. Some sub-sections could include:

Unit testing plans

The logging utility will be built as a DLL (otlog.dll). We will provide a binary otlog-test.exe which will exercise each of the interfaces mentioned above.

Stress testing plans

Use the unit testing executable in a mode where the logging files are overflowed etc.

Coverage testing plans

Cross-component testing plans

Upgrading/Supportability/Deployment design

Open Issues

eStream 1.0 Low Level Design

Software License And Management (SLiM) Server

Amit Patel
Last Modified: [redacted]
Version 2.0

Functionality

The Software License Management (SLiM) server is required to enforce licensing terms and track overall application usage. Its primary function is to grant, renew and expire access tokens, record application usage and aid in server load balancing. Its design can be broken down into three somewhat orthogonal axis:

1. Detailed specification of eStream client interfaces – the need.
2. Design and usage of Server Common Services (CSC) & server database –the tools of the trade. These include logging, system monitoring, thread package, encryption, TCP/IP communications, etc.
3. Core SLiM server logic that fulfils client interfaces (1) using CSC (2).

This document address items 1 and 3 in detail; item 2 should be covered in various other documents emerging from the server team.

Data type definitions

Common Data Types

- Standard atomic data types everyone (clients, builders, servers) must agree on: **Int8, Int16, Int32, Int64, uInt8, uInt16, uInt32, uInt64, uInt128** (specially for GUIDs).
  - Notice: No floating point types; I don't see a compelling reason to pass floating-point number across wire.
- **String** is represented as a size (uInt32) and it contents. Its contents must include a NULL termination character, it must be included as part of the size field.

Length	characters.....

- All eStream sets will use little endian representation.
- All complex data structures between major components (def: at least client/servers) should be version identifiable. Proposal: put a version number (uint32) as 1st word in the structure.
- Most complex structures are variable length because they contain strings. I think it would be good to put the total size (in bytes) of that structure as second word so that reader can know how much to read. If marshalling/unmarshalling utilities provide a way to represent that, it won't be needed in each struct.
- We'll need a single place where all globally (definition: across client and servers, and between servers) visible macros (#define & enum) are defined. We'll also need reserved

name spaces, and reserved number ranges for different components. Until all that is decided, I am defining macros without assigning any values.

- I am assuming that our message pack/unpack utilities will deal with alignment issues.

Server Sets

EstreamServerID contains server parameters clients need to know in order to initiate a connection. EStreamServerSet is simply a list of individual server Ids. The main use of this data structure is for SLiM server to return a list of app servers that can serve a given application. Server ids and server sets are specific to each application; client is responsible for keeping a map of app id → server set.

```
#define     SERVER_TYPE_APPLICATION
#define     SERVER_TYPE_SLIM
#define     SERVER_TYPE_ASP_WEB

typedef struct {
     uInt32    Version;
     uInt32    SizeInBytes;
     uInt32    MachineIP;
     uInt16    MachinePort;
     String    MachineName;
} eStreamServerID;

example of an eStreamServerID:
    {1,20,0x12348,80,{12,"s10.asp.com"}}

typedef struct {
     uInt32              Version,
     uInt32              SizeInBytes,
     uInt32              ServerType;
     uInt32              ServerIDCount,
     eStreamServerID     ServerID[];
} eStreamServerSet;

example of an eStreamServerSet:
{1,??, SERVER_TYPE_APPLICATION, 2,
     {1,20,0x12348,80,{12,"a10.asp.com"}},
     {1,20,0x12349,80,{12,"a11.asp.com"}}}
```

Access Tokens

This is a main data structure that is getting passed back and forth between a client and SLiM/App servers. Granting an AccessToken is an acknowledgement of client's legal right to run the application – the license. Denying an AccessToken is an acknowledging that the client does not have rights to run the application; probable causes include a user running multiple sessions, user not paying bills etc.

From client's perspective, it is totally opaque; but SLiM server uses it to pass information to the app servers so that the app server does not have to rely on the database lookups. Each access token will have a unique ID.

Terminology

Billing Granularity – Granularity at which an ASP is interested in billing its customers. Most ASPs today bill on monthly basis and eStream will assume that to be the 'norm'. However, to support things like short trial memberships, we'll design eStream to handle billing as often the AccessToken Renewal Frequency (defined below). If an end user simply purchases the eStreamed application, the billing granularity is infinite – the upper bound. EStream should not assume that billing granularity for all apps served by an ASP is the same.

AccessToken Renewal Frequency – Frequency at which the client must renew its access tokens in order to continue eStream application use. This must be tunable parameter whose upper bound is the billing granularity; it is also the smallest billing granularity we'll support. Not all access tokens are required to have the same renewal frequency.
Recommendation: **10 minutes.**
Tradeoffs:

1. This is the smallest granularity at which a client can be evicted (defined below).
2. Finer granularity may increase the number of hits to the SLiM server and adversely effect its scalability.

Eviction Notice – In general there will be times when an ASP wants to stop a user from using an eStream application, which also means stopping a user from consuming ASP server resources. Possible reasons may be:

- Lack of payment.
- Termination of a trial membership.
- To force the client into upgrading an app.
- Just because the restroom is freezing cold.

EStream infrastructure has an inherent limitation that servers can't push anything on the client. That means SLiM servers must deny an access token or its renewal, to effectively deliver an eviction notice to the client. Also, App servers may need to be informed of such evicted access tokens so that they can deny paging requests.

Decision: *After looking at some scalability numbers, we concluded that a renewal frequency of 10 minutes should not affect the overall performance and scalability of eStream system. Consequently, we don't have to communicate the list of evicted tokens to the app server since they would be invalid soon (avg 5 minutes) anyways. This simplifies server designs by reducing cross communication between slim servers and app servers.*

```
typedef struct {
  uInt32      Version;
  uInt32      SizeInBytes;
  uInt128     ATID;          // AccessToken ID - GUID
  String      UserId;        // GUID.
```

```
    uInt128      AppID;          // GUID.
    uInt64       IssueTime;      // POSIX time_t format.
    uInt64       ExpirationTime;
} eStreamAccessToken;
```

Other Common Data Structures

In order to allow easy/automatic updates of eStream application, we need to define a protocol by which a client can be informed of app updates. This structure will also be used when installing subscribed applications on a client.

AppName, VersionName – describe the application.
Message – a short description of an application.
Flages, such as ForcedUpgrade – client must upgrade the application.
RootFileNumber - is sort of the version # of an application root directory.
RootFileMetadata - metadata of the root directory.

```
typedef struct {
     uInt32     Version;
     uInt32     SizeInBytes;
     uInt128    AppID;
     String     AppName;        // may be "Word2000"
     String     VersionName;    // may be "SP1"
     String     Message;
     uInt8      ForcedUpgrade;
     Int32      RootFileNumber;
     ????       RootFileMetadata;

} eStreamAppInfo;
```

Interface definitions

In a single process context, cross-module interfaces are easy and intuitive when defined as C/C++ procedure calls. However, for client/server (and perhaps server/server) interfaces, we need to define our own RPC-like protocol. Sameer covering this (EMS – estream messaging services) in a different design, but I want to state couple of assumptions I am making:

- Each EMS call is assigned a unique number (Int32). Codes must be uniform across all servers (i.e. no duplication of names and numbers). We should reserve some namespaces and numbers for each eStream server. Following is the current list of EMS codes between SLiM/Clients.

```
#define     EMCC_NULL
#define     EMCC_ACQUIRE_ACCESS_TOKEN
#define     EMCC_RENEW_ACCESS_TOKEN
#define     EMCC_RELEASE_ACCESS_TOKEN
#define     EMCC_REFRESH_SERVER_SET
#define     EMCC_GET_LATEST_APP_INFO
#define     EMCC_GET_SUBSCRIPTION_LIST
```

- In addition to any data (pages etc.), an EMS calls needs to return a number of codes to communicate success/errors. Following structure provides a container for returning multiple return codes. By convention, we'll put either EMCR_FAILURE or EMCR_SUCCESS in Code[0].

```
typedef struct {
        uInt32          SizeInBytes;
        uInt32          ReturnCodeCount;
        uInt32          ReturnCodes[];
} EMCReturnCodes;

#define     EMCR_SUCCESS
#define     EMCR_FAILURE
#define     EMCR_USER_AUTH_FAILED
#define     EMCR_ACCESS_TOKEN_INVALID
#define     EMCR_SUBSCRIPTION_INVALID
#define     EMCR_LICENSE_NOT_AVAILABLE
#define     EMCR_LICENSE_ALREADY_HELD

#define     EMCR_EVICTION_NOTICE
#define     EMCR_EVICTION_MUST_UPGRADE
#define     EMCR_EVICTION_END_MEMBERSHIP
#define     EMCR_EVICTION_NO_PAYMENT
```

Acquire Access Token

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_ACQUIRE_ACCESS_TOKEN	
IN	uInt128	SubscriptionID
IN	String	UserName
IN	String	Password
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamAccessToken	AccessToken
OUT	uInt32	RenewalFreq
OUT	eStreamServerSet	AppServerSet
OUT	eStreamAppInfo	LatestAppInfo

Client will use this interface prior to starting an eStream application to grab the license. It accepts a subscription id, which clients received when an app was subscribed, and password; it replies with at least a list of return codes and possibly, the access token, its renewal frequency and a set of servers that can serve this app.
AccessToken – Client treats them as opaque data structures and renews them within its renewal frequency.

RenewalFreq – This uInt64 is the number of seconds the access token is valid for once it receives it. You probably don't want an absolute count (i.e. # of seconds since epoch) since clients can interpret it differently due to clock skew.
ServerSet – When a client gets an access token, it will be given a list of app servers that can serve the particular app. The ServerType member of eStreamServerSet structure will be SERVER_TYPE_APP. The list is specific to each app and should be managed as such.
LatestAppInfo – SliM servers will pass information about the latest app version (root) using this structure. Refer to client eFS design for more detail. This structure will always be passed; client will ignore it if it already has the latest version.

> Note: There is a big difference between major and minor upgrades: a major upgrade would be going from word 98 to work2000 (where app ids must change) where as a minor upgrade (app ids will not change) means applying a patch or a service pack. LatestAppInfo tries to transparently propagate latter (minor upgrades) to end users without requiring end users to unsubscribe/subscribe apps. Major upgrades will require end users to go back to the ASP web server and change subscriptions. ASP can force the end user into changing subscriptions (word 98 to word 2000) using EMCR_EVICTION_MUST_UPGRADE error code.

ReturnCodes
Success: EMCR_SUCCESS
Failure: EMCR_FAILURE, plus one of following:

- EMCR_USER_AUTH_FAILED – Can't authenticate user with specified passwd.
- EMCR_LICENSE_NOT_AVAILABLE – License is not available.
- EMCR_LICENSE_ALREADY_HELD – If the user is already holding the license, SLiM server returns this error code along with the access token that is held & its renewal frequency. Most common cause of this error is when an end user tries to run an eStream app on two different machines simultaneously. NOTE: returned token doesn't give the right to run the application and should be treated as a denial of access token. Reason for returning the token/renewal interval is to allow the client software can effectively release the token, wait some time (>= renewal frequency) and re-try.
  - The reason client has to wait is because SLiM servers will not communicate the list of 'bad' access tokens to the app server.
- EMCR_EVICTION_NOTICE –ASP wants to stop the user from using ASP resources. Server may also add code that describe the reason like 'no payment' etc. Note that no access token will be given! This may change in the future to allow some grace period.
  - EMCR_EVICTION_MUST_UPGRADE – This type of eviction means the ASP wants the end user to stop using this particular application in favor of another (major) version of it. For example, Word 98 to word 2000.

Renew Access Token

Caller:	eStream Client	
Callee:	SLiM Server	
RPC Code:	RPCC_RENEW_ACCESS_TOKEN	
IN	String	UserName
IN	String	Password

IN/OUT	eStreamAccessToken	AccessToken
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamServerSet	AppServerSet
OUT	uInt32	RenewalFreq

Clients will use this interface to renew the access token before it expires. Client will specify the old access token and if there are no errors, get back EMCR_SUCCESS, a new access token, new app server set (ServerType field of eStreamServerSet structure will be SERVER_TYPE_APP) and new renewal frequency. Upon getting the new app server set, client **must** remove the old app server set for this application. If for some reason, the access token is expired, SliM server will treat this request as 'Acquire Access Token' and may return error codes possibly from that interface (this is one of the reason for asking for usernames/password).

NOTE: Unlike Acquire Access Token, it is not returning LatestAppInfo or EMCR_EVICTION_MUST_UPGRADE error codes because once the app is running, we can't upgrade apps while running.

ReturnCodes

Success: EMCR_SUCCESS
Failure: EMCR_FAILURE, plus one of following:

- EMCR_ACCESS_TOKEN_INVALID – Access token is invalid.
- EMCR_EVICTION_NOTICE –ASP wants to stop the user from using ASP resources. Server may also add code that describe the reason like 'no payment' etc.
    - NOTE: will NOT return EMCR_EVICTION_MUST_UPGRADE.
- Error codes from Acquire Access Token.

Release Access Token

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_RELEASE_ACCESS_TOKEN	
IN	eStreamAccessToken	AccessToken
IN	String	UserName
IN	String	Password
OUT	EMCReturnCodes	ReturnCodes

Client will use this interface to release the license held by the specified user. It should be called synchronously when the application exits or crashes. The reason for requiring usernames and password is to authenticate the identity of the caller against access token owner. The reason for proactively releasing tokens as opposed to just letting them expire is because releasing it allows the user to re-acquire it (on the same or different machines) without waiting for it to expire. This allows the user to do acquire -> release -> acquire without any wait.

ReturnCodes

Success: EMCR_SUCCESS

Failure: EMCR_FAILURE, plus one of following:

- EMCR_ACCESS_TOKEN_INVALID
- EMCR_USER_AUTH_FAILED

Refresh App Server Set

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_REFRESH_APP_SERVER_SET	
IN	eStreamAccessToken	AccessToken
IN	uInt8	BadQoS
IN	uInt8	NoService
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamServerSet	ServerSet

App Server sets are given to a client when an access token is acquired and are automatically refreshed when an access token is renewed. However, the client can always refresh its app server sets using this interface. Potential reasons for clients to do this:

- All servers in the current server set are not responsive – NoService = TRUE
- Servers are up, but client experiences bad QoS (network delays/timouts). BadQoS = TRUE

ReturnCodes

Success: EMCR_SUCCESS

Failure: EMCR_FAILURE, plus one of following:

- EMCR_ACCESS_TOKEN_INVALID

Get Subscription List

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	EMCC_GET_SUBSCRIPTION_LIST	
IN	String	UserName
IN	String	Password
OUT	EMCReturnCodes	ReturnCodes
OUT	uInt32	NumberOfSubscriptions
OUT	uInt128[]	SubscriptionID[]

A client can ask for the current list of subscribed applications using this interface. SLiM server returns the number of apps subscribed and an array of subscription ids.

ReturnCodes

Success: EMCR_SUCCESS

Failure: EMCR_FAILURE, plus one of following:

- EMCR_USER_AUTH_FAILED

Get Latest Application Info

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_LATEST_APP_INFO	
IN	String	UserName
IN	String	Password
IN	uInt128	SubscriptionID
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamAppInfo	UpgradeInfo

Any upgrades pending? This functionality is piggy backed on 'acquire access token' interface, but there is some value in providing it as an explicit interface. SliM server will give you the latest application information block associated with the specified subscription id; the client can decide if it already has the latest root (version) or not.

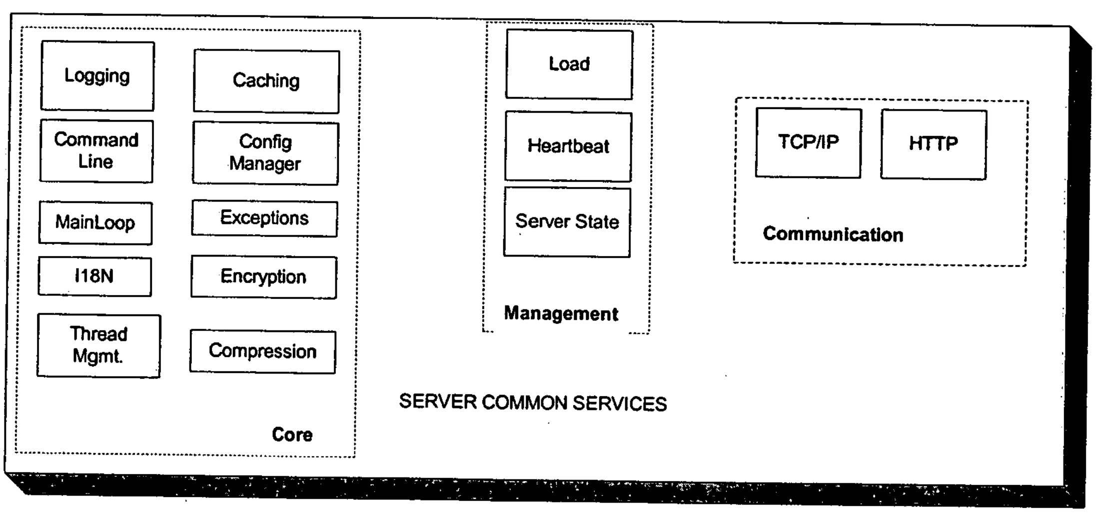
ReturnCodes
Success: EMCR_SUCCESS
Failure: EMCR_FAILURE, plus one of following:

- EMCR_USER_AUTH_FAILED
- EMCR_SUBSCRIPTION_INVALID

Component design

Server Common Services

Following diagram shows the common portion of all eStream servers. Most of these boxes won't be described in this document because they are covered in specific documents.



Various Decisions

- SLiM server will uses an ODBC interface to communicate with the central database.

- From eStream client's perspective, all SLiM servers are equal in functionality. This is unlike an application server, which can be segmented to serve specific applications.
- Each SLiM server will have a unique IP/Port combination. Multiple SLiM servers running on the same machine can be distinguished by giving them different port numbers.
- SLiM servers (and app servers) will not assume any default ports; it will rely on an ASP admin to configure the port assignment. With help from OTI, ASP admin will determine how many eStream servers need to run on a machine and assign a unique number to each eStream server.

Hardware Failover

There will be a pool of SLiM servers at an ASP site; from eStream client's prospective, each of them is identical. When a client subscribes to an eStream application, it gets a set of SLiM servers to communicate with. The clients will keep this list in memory and refer to it when calling SLiM server interfaces. If it experiences difficulty communicating with a particular SliM server, it will try other servers that are part of the server set. If for some reason the server set is lost, or all servers in the set are not responding, a client can always go back to the ASP web server and refresh its server set. This gives you a transparent (from end user's prospective) hardware fail over path.

The same approach will also work for app server fail-over scenarios; specific differences are that:

1. SLiM Servers, not ASP web servers, will provide the app server set.
2. App server list will be refreshed automatically, when access tokens are renewed. This allows ASP admins to take out servers from the pool by waiting certain amount of time (>= access token renewal frequency) and not cause unnecessary client timeouts.
3. App server sets are specific to each app; SLiM servers are not.

Load Balancing

In eStream 1.0, we will not require a third party load balancers at an ASP site; we'll do minimal things at both ends (clients/server) that should be good enough for small to medium size ASPs. We may have to test with selective 3rd party load balancers to see if we can work with them or not; but this is an open issue (listed in the open issues at the end of doc.).

In eStream 1.0, we'll capitalize on the hardware fail-over mechanism to also aid load balancing. Following two actions will perform load balancing:

- When a client gets server sets (app or SLiM servers), it will distribute its hits randomly among the server in the set. In addition, clients will also get new app server sets every time they renew access tokens.
- On server side, the monitor will keep track of each server's response times to process client's requests. The data gets sorted from most responsive to least responsive and stored into the database. Top 'X' servers from this list will be given to the client when it makes an explicit request to refresh its app server set, acquires or renews an access token.

Testing design

Interface Testing

SLiM server is tightly coupled with three components: client, ODBC/Database and monitor; it is fairly difficult to isolate it from *all* of those components for unit testing. A better approach is to exhaustively test the client/SLiM server interfaces, which will in fact also test large numbers of interfaces to the other two components. The idea is to crank up a client that will make every possible SLiM server request and make sure that SLiM server responds accordingly.

I think it is good idea to create a simple testing framework (that may evolve with time) that will simulate a real client to SLiM and app servers. We can do this by writing a program that includes common (client/server) data structures definitions, links in our eStream Message Services (EMS) component and invokes various interfaces like 'Acquire Access Token'. From SLiM server's prospective, this test program is a working client.

For each client/server interface (i.e. Acquire access token) write a test case (dummy client) that will:

- Assume that we have created a dummy database that has certain users, passwords and subscriptions.
- Invoke SLiM server with all possible input permutations. This isn't too bad since most interfaces have 2 to 4 arguments.
- In the process, ensure that SLiM server returns all possible return values it can.

  For instance, lets assume that Acquire Access Token has following prototype:
  AET(uInt128 subID, String UserName, String Password);

  TEST BEGIN:
  Assert (AET(NULL, NULL, NULL) returns
  EMCR_FAILURE & EMCR_USER_AUTH_FAILED);

  Assert (AET(good_sub_id, good_user_name, good_password) returns
  EMCR_SUCCESS, an access token, its renewal freq. Etc.);

  Assert (AET(good_sub_id, good_user_name, good_password) returns
  EMCR_FAILURE & EMCR_LICENSE_ALREADY_HELD);

Stress testing plans

Stress testing in general will be common across all eStream servers. I think it will be a good idea to invest in a 3rd party tool that can simulate real-time load on eStream servers and see its responses. Rational has various tools such as Visual Test, Robot and Site Load that are worth evaluating.

Coverage testing plans

Lot of these items will apply to all eStream components and probably should be covered in a separate eStream test plan document. I am not sure if we should do these things before each component is done or wait until they are integrated. I just want to state what may be obvious so that it is documented:

- SLiM server will achieve 85% PFA coverage as measured by Rational PureCov. Tests used to measure PFA coverage will be reproducible, either by hand or via an automated test suite.
- SLiM server will resolve all memory corruption and memory leak issues as reported by Rational Purify.
- We should have test cases that will exercise all command line options for SLiM server.
- SLiM server will be code reviewed by at least two peers.

Upgrading/Supportability/Deployment design

This document must have a discussion of how the component addresses any specific issues related to upgrading, supporting and deployment of e-stream applications. Some examples include: error conditions detected and reported by this component, any special hooks this component will provide for monitoring, hints for troubleshooting problems, any special hooks for debugging this component.

Open Issues

This is a list of issues that need to be further investigated or revisited during implementation.

1. How do you produce GUIDs on unix servers? Should app ids, user ids, access token ids be guids or we should create them by knowing what numbers are already used?
2. Resolve big-endian – little-endian issues. Owner: Sameer
3. Meaning of 'eviction' notice is not conveyed to the end user yet. Owner: Client person – Ann.
4. Encryption impact on SLiM servers. Owners: Amit & Igor.
5. Global name space & number ranges for different components. Owner: Bhaven
6. ASCII v/s Unicode strings? Owner: Sameer.
7. Test with 3rd party load balancers to see if we work or not. Requirements for deployment team: tell us which load balancer to certify against and set them up in our future testing lab. Owner: deployment team.

eStream Web Server/Database Low Level Design

Bhaven Avalani
[redacted]
Modified: [redacted]

Functionality

The eStream solution provides a set of account, user, and subscription management utilities. These utilities are provided as extensions to the ASP's(Application Service Provider) web server.

There are three categories of users for these utilities: End User, Group Administrator and ASP Administrator. The roles and the capabilities of each of these users are detailed below.

End user for a system is the user who will actually access eStream application using the eStream clients. An end user should be able to:

- Create Account and User attributes. (Username, Password, etc.)
- Change Account and User attributes.
- View all available applications in the eStream system.
- Subscribe/Manage eStream applications.
- View Account Status.
    1. List of applications subscribed.
    2. Status of current subscription.
    3. View/Change Billing information.
    4. View/Change Account information.

A *Group Administrator* is an administrator for a group of users. An individual user is by definition a group administrator for a single user group. Capabilities of a group administrator are:

- (All of single user capabilities).
- Add delete users from a group.
- Manage the active sessions for a group. A group manager should be able to release licenses from active sessions, thereby kicking out active users.
- View the billing information. This will probably need hooks to an external billing system.

An *ASP administrator* manages the overall application system. Capabilities of an ASP administrator are:

- Manage accounts/users/subscription for all users/groups in the system.
- Manage the application data for a subscription system.

    1. Add new applications to the system.
    2. Modify application information for the system.
    3. Provide the pricing mechanism for the applications(?).
- Manage the servers in the system.
    1. Configure a server.
    2. Stop/Start a server. This is accomplished by a message to the Monitor server.
    3. Get load information for a server.
    4. Get logging information for a server.

There are essentially two different types of accounts, which the system will support: Single user account and corporate accounts.

The following licensing mechanisms will be supported by the system.

- Fixed Duration License. (Typically monthly license).
- Fixed Duration Floating License. An example of this is n licenses for k users for a fixed duration.
- Indefinite License.

Description

There are several key issues that need to be determined for the Web Server architecture. The options available in the market to implement these technologies are listed below.

Web Server:

- Apache
- Netscape Server
- Microsoft Internet Information Server

CGI Technology

- Servlet/JSP
    - Tomcat (from Apache group)
    - JRun (from Allaire)
- Active Server Pages (available on NT only)
- NSAPI (C level API available for Netscape and Apache).
- ISAPI (C level API available for IIS and Apache)
- CGI (Perl/C etc.).

Database Connectivity

- JDBC.

- ODBC
- Native.

Database

- SQLServer
- Oracle
- Sybase
- Informix
- LDAP(??)

The overall proposed solution for eStream 1.0 WebServer release is:

Apache + Tomcat(for JSP/Servlet) + JDBC + SQLServer.

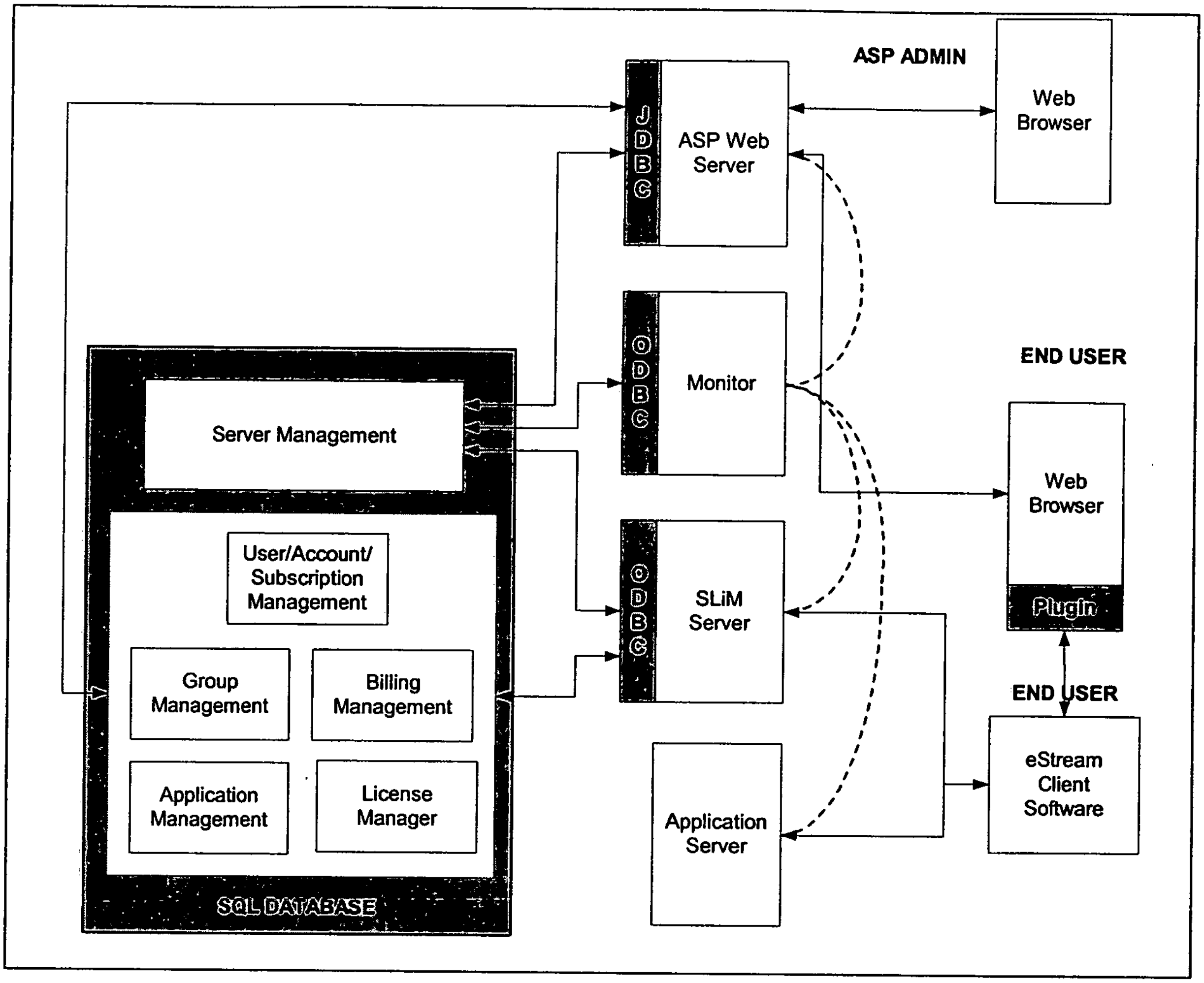
The reasons for choosing this combination for the servers are as follows:

1. JSP/Servlet is the only technology which is available for cross-platform and cross WebServer support.
2. We need to decide on a single web server to develop and test against for release 1.0. Apache is chosen to be the one as it is popular on Unix and NT platforms and it is freely available.
3. Tomcat(Apache group's reference implementation for JSP/Servlet specs) is the preferred CGI technology as it works well with Apache and all other web servers.
4. JDBC is preferred for database connectivity as its database neutral and works well with Java environment of Servlets.
5. SQLServer is the preferred database for release 1.0. This contains the scope for testing and deployment for eStream 1.0.

Since all other servers(App Server, SLM Server and Monitor) are C++ components, the following technology combination will be available for Database Access.

ODBC + SQLServer.

The data model for the eStream 1.0 database essentially consists of two high level components. The database deployment architecture is shown below:



Server Management Component: This component's primary responsibility is to manage the configuration, load and log information for a logical server in the system. The clients to this component are all the servers and administration manager. A detailed list of interfaces for this component is described in the interfaces section.

User/Account/Subscription Management: This component is responsible for maintaining the user account and subscription information for the system. The end user using the end user interface performs the updates to this component. Slim Server will access this component to validate subscriptions. A detailed list of interfaces for this component is described in the interfaces section.

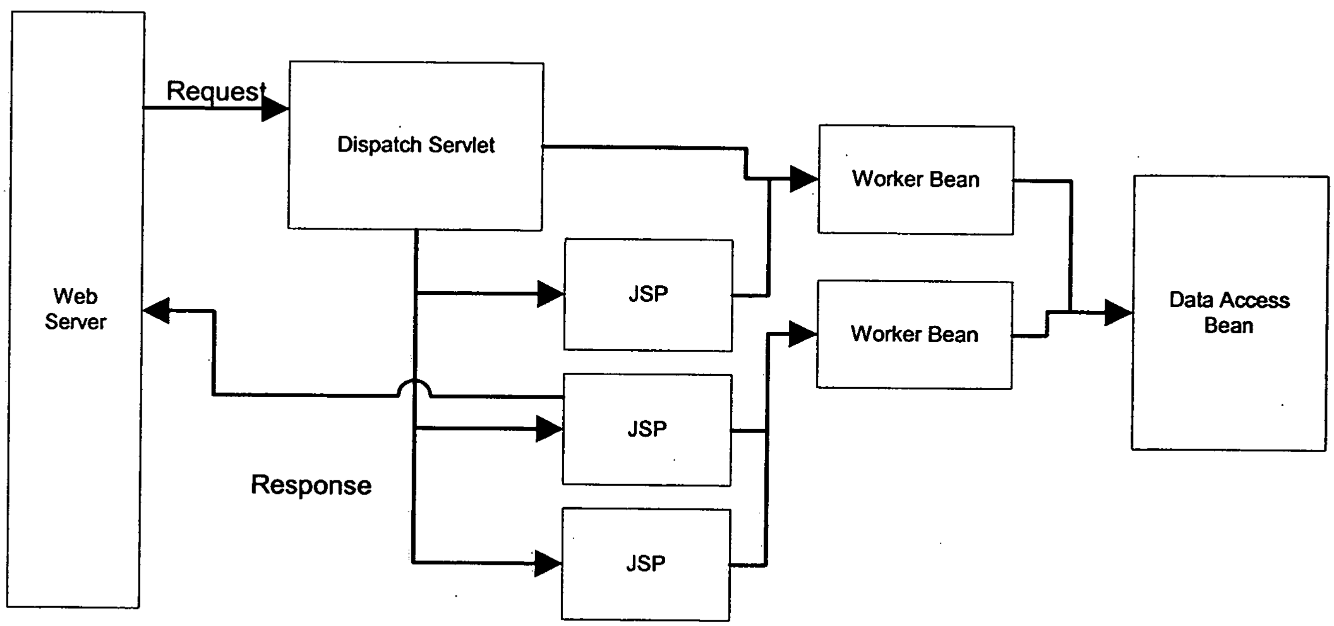
Group Management: This component is useful for managing groups of users. The group administrator can only perform updates to this component. A detailed list of interfaces for this component is described in the interfaces section.

Billing Management: This component's responsibility is to provide interfaces to an external billing system. A detailed list of interfaces for this component is described in the interfaces section.

Application Management: This component's responsibility is to provide application management interface. This component is accessible for updates only by the ASP administrator. A detailed list of interfaces for this component is described in the interfaces section.

License Manager: This component's responsibility is to manage the licenses. SLiM server will check out licenses from the license manager. A detailed list of interfaces for this component is described in the interfaces section.

The architecture for the Web Server extensions implementation is shown below:



The basic elements of this architecture are as follows:

1. Every request into the system goes through a dispatcher servlet. This servlet will perform initialization, initial validation of the request and miscellaneous checks before dispatching the request to a JSP page. A worker bean will responsible for performing the initialization. The processing of the incoming request is performed at this stage. The request is then dispatched to an appropriate JSP page.
2. The JSP page will invoke worker beans to access the dynamic data from the database via the Data Access Bean and the resultant page is sent back to the user.

This architecture is illustrated with the following example.

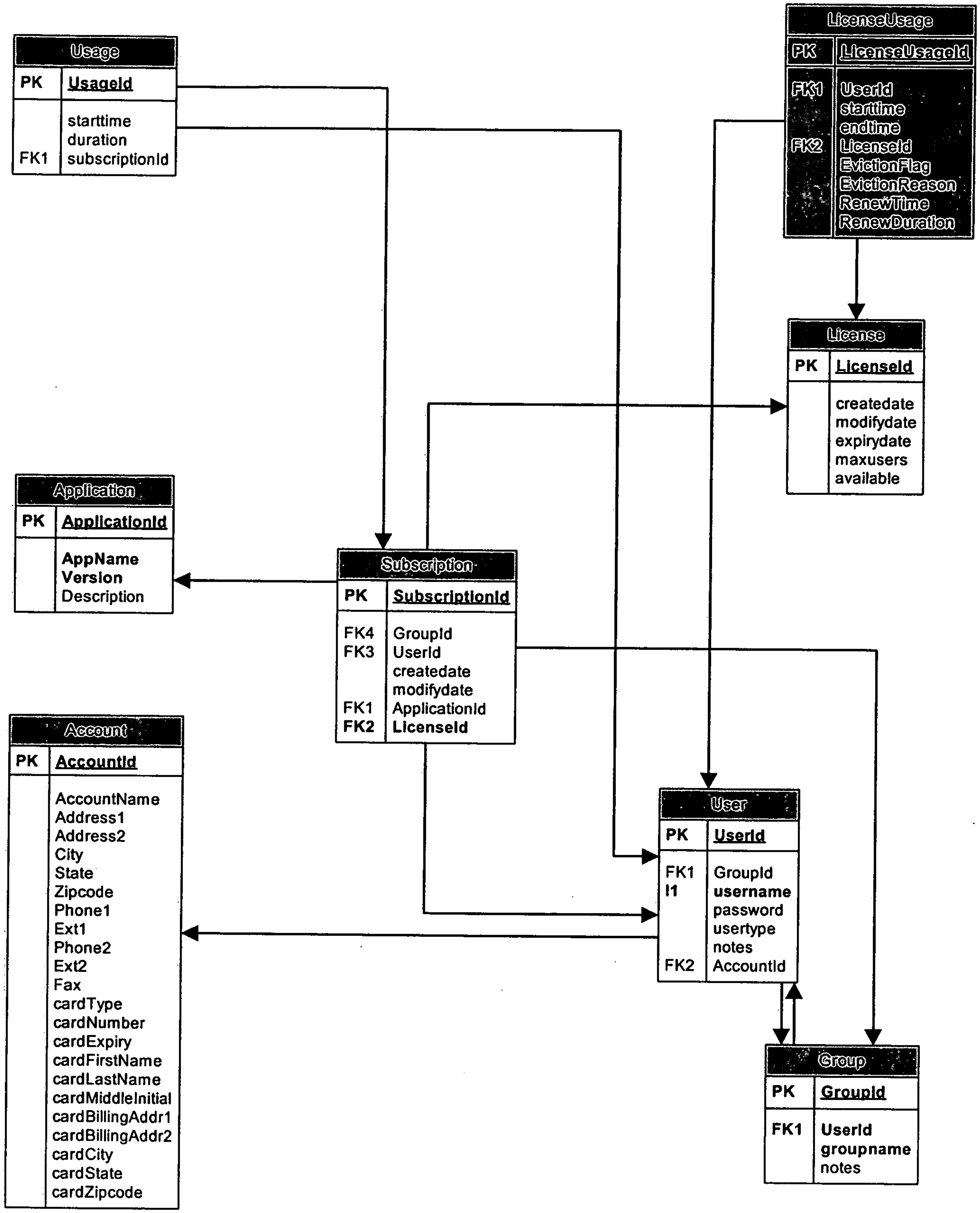
1. User sends in a request to update the username and password information in the database. Inputs are username, old password, new password.
2. The dispatch bean will call the user(worker) bean to:
   a. Validate the user's old password.
      i. The user worker bean will make a request to the data access bean to access the password for the user.
      ii. The two passwords are compared and the result is returned.
   b. If the password was valid then, update the new password.
      i. Call the data access bean to update the password in the database.
   c. Else return failure.
3. Based on the success or failure the dispatcher will dispatch the page request to the appropriate JSP page. (eg. error.jsp on failure and user.jsp on success).
4. The page will invoke the appropriate the worker bean (error bean or user bean) to obtain the dynamic data and send the response back to the user.

The salient features of this architecture are:

1. Presentation and processing logic is separate. Thus, the customer(ASP) can customize the look and the feel of the pages without impacting the processing logic as it is segregated.
2. The data access bean is separated from the worker beans, which are primarily responsible for the business logic. This allows us to change the data access layer (eg enabling LDAP access) in the future without impacting the system drastically.

Data type definitions

The central data structure for Web Server is the database model. The overall database model for user and subscription management is shown below.



Usage
PK UsageId
starttime
duration
FK1 subscriptionId
LicenseUsage
PK LicenseUsageId
FK1 UserId
starttime
endtime
FK2 LicenseId
EvictionFlag
EvictionReason
RenewTime
RenewDuration
License
PK LicenseId
createdate
modifydate
expirydate
maxusers
available
Application
PK ApplicationId
AppName
Version
Description
Subscription
PK SubscriptionId
FK4 GroupId
FK3 UserId
createdate
modifydate
FK1 ApplicationId
FK2 LicenseId
Account
PK AccountId
AccountName
Address1
Address2
City
State
Zipcode
Phone1
Ext1
Phone2
Ext2
Fax
cardType
cardNumber
cardExpiry
cardFirstName
cardLastName
cardMiddleInitial
cardBillingAddr1
cardBillingAddr2
cardCity
cardState
cardZipcode
User
PK UserId
FK1 GroupId
I1 username
password
usertype
notes
FK2 AccountId
Group
PK GroupId
FK1 UserId
groupname
notes

The important features of this data model are:

Account: Table holding all the billing and contact information for a user or a group.

User: An end user in the system. A user can optionally belong to a group.

Group: A group of users. One of the users in the group is designated as the group administrator. Each group has a unique account associated with it.

Application: This table contains the data about various applications in the supported by the ASP.

License: Each row in this table corresponds to the licensing term for a given subscription. This table also maintains the active count of the licenses checked out.

Subscription: This table contains entries for subscription items. A subscription item consists of user/group, application and license.

Usage: This table contains the runtime information for a system. SLiM server updates this table with access token usage data. A billing system may interface with this table to generate billing data. A reporting system may interface with this table to report on usage patterns.

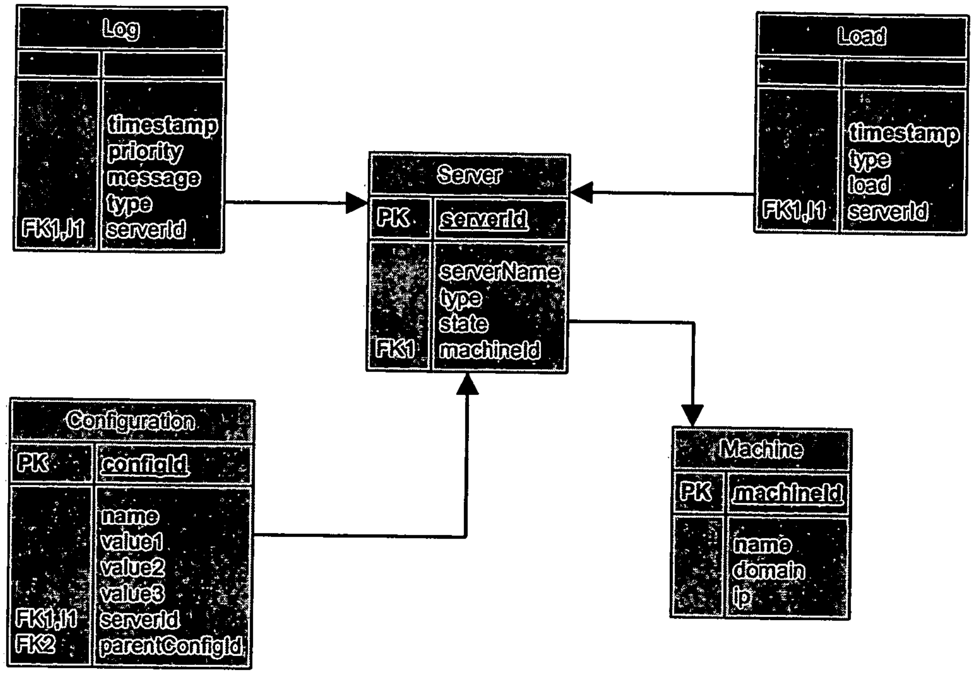
LicenseUsage: This table is responsible for recording checked out licenses in the system.

The data model for storing the server related information is shown below:

PK: Primary Key for the table.

FK: Foreign Key. Used for relations between tables.

I1,I2.. : Index Columns.



The tables in this model are:

Server: This table contains entries for each logical server in the system.

Machine: This table contains entries for each physical server in the system

Configuration: This table contains configuration entries for a given server. The configuration entries can be hierarchical in nature. Each configuration has the following format:

Name Value1 [Value2] [Value3] [ParentConfigId]

Load: This table maintains the historical and real-time load information for a given logical server in the system.

Log: This table maintains the logs for a logical server in the system. The log messages saved here are "major" events in the logical server system. A detailed logs stored in a flat file on the physical machine containing the logical servers.

Global Data Structures:

```
struct ServerTuple
{
int serverId,
int type,
String serverName
};
```

```
struct Couple
{
String name,
String value
};
```

For the Access Token and related data structures, please refer to the SLiM server Low Level Design Document. The interfaces below will discuss some of the API's based on the these data structures.

Interface definitions

The interfaces exposed by various sub-components are detailed below.

Server Management Component:

CreateServer

int CreateServer (ServerConfig* config)

```
Input:
   Server Configuration.
Output:
   Unique ID for the server.
Comments:
   Create a logical server with predefined con-
figuration.
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

UpdateServerConfig

Bool UpdateServerConfig(int serverId, String name, String value)

```
Input:
   Server Id
   Config name and value
Output:
   Unique ID for the server.
Comments:
   Create a logical server with predefined con-
figuration.
```

Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

AddMachine

Bool AddMachine(String name, String domain, String ip)

Input:
Machine name,domain and ip.
Output:
Success/Failure
Comments:
Create a physical machine entry.
Errors:
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

SetServerLog

Bool SetServerLog(int serverId, LogTuple log)

Input:
Server Id
Log tuple(data structure in the Logging document.)
Output:
Success/Failure
Comments:
Add the log data for a server
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServerLog

LogTuple[] GetServerLog(int serverId, int maxrows = 25)

Input:
Server Id
Maxrows: Maximum number of rows to be returned.

Output:
Array of Log tuples(data structure in the Logging document.)
Comments:
Get the log data for a server
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServers

Server[] GetServers()

Input:

Output:
Array of Server tuples(data structure defined above)
Comments:
Get all the server information
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

SetServerState

Bool SetServerState (int serverId, short state)

Input:
ServerId: Unique id for a server
State: State information for a server.
Output:
Bool True/False for success/failure.
Comments:
Update the database with current state information for a specified server
Errors:
INVALID SERVER ID
DB ROW LOCKED
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServerState: Obtain the last known state for a specified server

short GetServerState (int serverId)

```
Input:
   ServerId: Unique id for a server
Output:
   State: State information for a server.
Comments:
   Obtain the last known state for a specified
server
Errors:
   INVALID SERVER ID
   DB ROW LOCKED
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerConfig: Obtain configuration information for a specified server

ServerConfig* GetServerConfig (int serverId)

```
Input:
   ServerId: Unique id for a server
Output:
   ServerConfig*: State information for a
server. (ServerConfig data structure is defined
in the server configuration document).
Comments:
   Obtain the last known state for a specified
server
Errors:
   INVALID SERVER ID
   DB ROW LOCKED
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

SetLoadData:

void SetLoadData (int serverId, int Load)

```
Input:
   ServerId: Unique id for a server
   Load: Load for the server
```

Output:

Comments:
Monitor may call this interface to persistently store historical load data. It is still not clear if SLM and application servers will store this directly themselves.
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServerLoad:

void GetServerLoad (int serverId, , int maxrows = 25, int** Load)

Input:
ServerId: Unique id for a server
maxrows: Maximum number of rows to be returned.Default is 25.
Output:
Load: Load for the server
Comments:
Obtain server component load information to manage load balance.
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

FlushLoadData:

void FlushLoadData (<tuples> LoadData)

Input:
LoadData tuples containing <server id, server load> values.
Output:

Comments:
Used to flush aggregated load data to the databa

Errors:
```
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

User/Account/Subscription Management Component

CreateUser. This API is used to create user record in the system. Arguments will be Username, Password.

Bool CreateUser(String username, String password)

ValidateUser. This API is used to validate user record in the system. Arguments will be Username, Password.

Bool ValidateUser(String username, String password)

CreateAccount. This API is used to create account records in the system. Arguments will be billing address, credit card information etc.

Bool CreateAccount(String username, <Account Information>couple[])
Input:
Username associated with the account.
An array of names and values for the account.

AddSubscription. This API is used by the end users/group administrators to subscribe to applications.

Bool AddSubscription(<Subscription Information>couple[])
Input: An array of names and values for the subscription.

UpdatePassword. Used to change user information. Password, username etc.

Bool UpdatePassword(String username, String old-password, String new-password);

UpdateAccount. Used to update the account information. Billing Address etc.

Bool UpdateAccount(Couple[])

Input: An array of name, value pairs for the fields to be updated.

UpdateSubscription. Used to add additional time to a subscription.

Bool UpdateAccount(Couple[])

Input: An array of name, value pairs for the fields to be updated.

GetUserRecord. Used to get current user configuration.

Couple[] GetUserRecord (String username)

GetAccountRecord. Used to get current account configuration for a user.

Couple[] GetAccountRecord(String username)

GetSubscriptionRecords. Used to get to subscription records in a database. End user may just want to verify what they are subscribed to.

Couple[][] GetSubscriptionRecords(String username)

Output: An array of array of couples containing the subscription information for a given user.

DeleteUser. Used to delete users who are no longer valid in the system. Typically called by the ASP admin.

Bool DeleteUser(String username)

DeleteAccount. Used to delete un-used accounts.

Bool DeleteAccount(int accountId)

DeleteSubscription. Used by the ASP admin to remove subscriptions.

Bool DeleteSubscription(int subscriptionId)

Group Management Component

CreateGroup. This API is responsible for creating group accounts in the database. Called by the group admin user.

Bool CreateGroup(String groupName, String admin, String notes)

AddUserToGroup. Adds a user to a group.

Bool AddUserToGroup(String groupName, String username)

DeleteUserFromGroup. Removes a user from a group.

Bool DeleteUserFromGroup(String groupName, String username)

GetActiveSessions. Gets the active sessions for a group.

Couple[][] GetActiveSessions(String groupName)

Output: An array of array of couples containing the following information for each active session in the system:

Username
LicenseId
StartTime
EndTime
Subscription

Licensing Component

CheckoutLicense: Checks out a license.

int CheckOutLicense(int subscriptionId, long* pStartTime, long* pStopTime)

Inputs:

SubscriptionId: Subscription id of the user.

Outputs:

>0 for a successful license. The output is the license usage id.
StartTime for the license.
StopTime for the license.

Comments:

The system should validate the availability of the license.

Errors:

```
INVALID USER ID
INVALID SUBSCRIPTION
LICENSE NOT AVAILABLE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

RefreshLicense: Refreshes a license.

int RefreshLicense(int LicenseUsageId, long* pStartTime, long* pStopTime)

Inputs:

LicenseUsageId: License usage id.

Outputs:

>0 for a successful license. The output is the license usage id.
StartTime for the license.
StopTime for the license.

Comments:

The system should validate the availability of the license.

Errors:

```
INVALID USER ID
INVALID SUBSCRIPTION
LICENSE NOT AVAILABLE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
EVICTION
```

CheckinLicense: Check in a license

Bool CheckInLicense(String username, int subscriptionId, int licenseUsageId)

Inputs:

Username: user trying to check out the license
SubscriptionId: Subscription id of the user.
LicenseUsageId: Usage id for the checked out license.

Outputs:

Success/Failure

Comments:

Errors:

```
INVALID SUBSCRIPTION
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

ValidateLicense: Validate that the user has a license checked out.

Bool ValidateLicense(String username, int subscriptionId, int licen-seUsageId)

Inputs:

Username: user trying to check out the license
SubscriptionId: Subscription id of the user.
LicenseUsageId: Usage id for the checked out license.

Outputs:

Yes/No.

Comments:

Errors:

```
INVALID USER
INVALID SUBSCRIPTION
INVALID LICENSE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

DBAcquireAccessToken

RPCReturnCodes DBAcquireAccessToken(long SubscriptionId, long* pAccessTokenId, string UserName, string Password, long* pStartTime, long* pStopTime, long* ApplicationId)

IN	SubscriptionId	Id of the subscription being used.
IN/OUT	pAccessTokenId	-1 if this is a first time access.
IN	UserName	Username string.
IN	PassWord	Encrypted Password
OUT	pStartTime	Start time for Access Token validity.
OUT	pStopTime	Stop time for Access Token validity.
IN/OUT	ApplicationId	Id of the application. –1 Default.
OUT	RPCReturnCodes	RPC Return codes.

Processing:

This is fairly complex function. The processing involved in this function call is:

- If this is the first access (ie *pAccessTokenId == -1) then **ValidateUser**
- If the ApplicationId is –1 then **GetAppId**
- If this is the first access (ie *pAccessTokenId == -1) then **CheckoutLicense**
- If this is a renewal request: **RefreshLicense**
- If there is a failure and it is due to eviction: **GetEvictionReason**

Errors:

```
#define RPCR_USER_AUTH_FAILED
#define RPCR_ACCESS_TOKEN_INVALID
#define RPCR_ACCESS_TOKEN_EXPIRED
#define RPCR_LICENSE_NOT_AVAILABLE
#define RPCR_LICENSE_ALREADY_HELD
#define RPCR_EVICTION_NOTICE
```

```
#define    RPCR_EVICTION_MUST_UPGRADE
#define    RPCR_EVICTION_END_MEMBERSHIP
#define    RPCR_EVICTION_NO_PAYMENT
```

DBReleaseAccessToken
DBReleaseAccessToken(long AccessTokenId)

IN   AccessTokenId

Processing:
- Update the Usage table with the appropriate information.
- Delete the LicenseUsage record.

Notes:
- We need a mechanism to release un-released access tokens. The way to do this would be to run a stored procedure at demand and at a predefined intervals to do this cleaning up.

EvictAccessToken
DBReleaseAccessToken(long AccessTokenId)

IN   AccessTokenId

Processing:
- Evicts an access token.

Billing Component

AddUsageRecord. Called by the SLM server when it releases an access token.

Bool AddUsageRecord(String username, int subscriptionId, date starttime, long duration).

GetUsageRecordsForUser. Used by external billing system.

Couple[][] GetUsageRecordsForUser(String username)

GetUsageRecordsForGroup Used by external billing system.

Couple[][]GetUsageRecordsForGroup (String groupName)

Application Management Component

AddApplication

int AddApplication(String appname, String version, String description)

Inputs:

Appaname: Application name.
Appversion: Application version
Description. Application description.

Outputs:

-1 for failure to add the application.
>0 otherwise. Application ID.

Comments:

Returns an app id for a newly added application.

Errors:

APPLICATION EXISTS
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetApplicationId

int GetApplicationId(String appname, int version)

Inputs:

Appaname: Application name.
Appversion: Application version

Outputs:

-1 for failure to find the application.
>0 otherwise.

Comments:

Returns an app id for a given application.

Errors:

NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetApplicationId

int GetApplicationId(int SubscriptionId)

Inputs:

SubscriptionId

Outputs:

0 for failure to find the application.

>0 otherwise.

Comments:

Returns an app id for a given application.

Errors:

```
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetSubscribedApplicationIds

Int[]* GetSubscribedApplicationId(String username)

Inputs:

Username: username.

Outputs:

Array of application ids subscribed by a user.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetUnsubscribedApplicationIds

Int[]* GetUnsubscribedApplicationId(String username)

Inputs:

Username: username.

Outputs:

Array of application ids not subscribed by a user.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetApplicationDetail

Couple[] GetApplicationDetail(int appid)

Inputs:

Application Id.

Outputs:

Array of couple for the app id containing:
{appname, appversion, description} values.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

Component design

We will discuss some complex scenarios in this section.

Subscription

Single New User

1. Create the user. **CreateUser.**
   a. If a user already exists, return error message and go back to 1.
2. Create the account for the user. **CreateAccount**
   a. Get the contact information from the user.
   b. Prompt to get the billing information. The user may decide to not give the billing information at this point.

Corporate group admin creating an account.

1. Create the admin user. **CreateUser.**
2. Create the group. **CreateGroup**
3. Create the account information for the group. **CreateAccount.**
   c. Get the contact information from the user.
   d. Prompt to get the billing information.
4. Add users to the group. **AddUserToGroup.**
   a. This method will automatically create the user if they do not already exist in the system.
   b. The list of users is accessible to the Group Admin by querying:
      i. Our database **GetUserRecords** OR
      ii. Some external database. Eg. LDAP directory.

Single User subscribing to an application

1. Validate the user. **ValidateUser**

2. Prompt to get the billing information if the billing information is not already present.
3. Get the list of un-subscribed applications. GetUnsubscribedApplications.
   a. **GetUnsubscribedApplicationIds.**
   b. For each app id returned, get the application details. **GetApplicationDetail**
4. For each additional application user wants to subscribe, call **AddSubscription**

SLiM server checking out an access token to use an application

1. Call **DBAcquireAccessToken.**

Testing design

This document must have a discussion of how the component is to be tested. Some subsections could include:

Unit testing plans

The follwowing components will be unit tested:

ODBC connectivity dll for the SliM server and the Monitor. A simple C++ executable will be provided to test the SLiM server and the Monitor interfaces. The C++ executable will:

- establish connection to the database.
- simulate access token calls.
- simulate the monitor calls.

The Web servers' servlets and the JSP pages interact with the database using a set of java beans. Each of the bean will have a test interface. A simple JSP will be provided which will call all of the test interfaces for the beans. The test interface itself will be responsible to call all the interfaces that the bean provides in a predefined calling sequence.

The servlets will be unit tested using a set of html forms which will invoke the servlets.

Stress testing plans

Stress testing will be invoked using some external testing tool which can record HTTP traffic and replay the traffic for multiple users. Performix and LoadRunner are two possible choices. (There may be additional tools available).

Coverage testing plans

Coverage on the ODBC components will use the same mechanism as the rest of the server code. Pure Coverage may be used to achieve this goal.

Coverage on the Java components is an open issue. We need to investigate the appropriate tools for doing this testing.

Cross-component testing plans

The database is the central point for distribution of the data from Web server to the rest of the servers. Thus, creating the database data which can be used by Slim server and the App server will be a good source of cross component testing plan.

Upgrading/Supportability/Deployment design

We will be finally shipping just the java class files and JSP pages to the customer. It is assumed that the customer will have the appropriate web server to support JSP 1.1 and Servlet 2.2.

Open Issues

1. We have assumed that the JDBC implementation will come from Inet software. We may need to change to an alternate JDBC vendor based on pricing, quality etc.
2. Tomcat is decided to be the JSP/Servlet engine. Again, this is a freeware and may be replaced by a commercial version (Jrun from Allaire).
3. The web server will need to talk to the Monitor. The messaging component for this communication is not well defined yet.

eStream 1.0 High Level Design

Version 0.3

[redacted]

Notes

The following is roughly what's changed since the last version (0.2):

- The functional requirements and use cases have been removed. These will be documented in the eStream Requirements Document in future revs.
- The entire accounting hierarchy (what is a user and account, how are they grouped, at what level does billing take place) is undergoing revision, and has been removed from here for this version.
- Component descriptions should be more consistent now.
- The database of user and subscription information in the client block diagram has been removed. See the notes below.

Known issues

- The mechanism for how a pathname on a client machine translates into a globally unique FileID for any eStream server is unclear. This is a major design issue that crosses many components on both the client and the servers.
- The accounting hierarchy and its impact on this design are missing.
- If and how copy-on-write will work for writes to the Z file system is quite unknown.
- Which server manages user/account/group/subscription data is quite uncertain. Representing this by a data cylinder was wrong, and I removed this. However, all the interfaces specified below for an "ASP web server" are now just plain wrong, and the server team needs to suggest the appropriate changes to the HLD for the server topology.
- The "Server Data Objects" section at the end of this document needs to be rewritten, in terms of interfaces that client and server components supply to support these data.

Introduction

This document describes the high level design for the eStream 1.0 product. The organization used is:

- Definitions
- Block diagrams for both the client and server portions, showing all major components
- Each component, generally broken down by
    - purpose

- functionality
- global data managed, if any
- interfaces for use by other components

To understand the problem being solved in this design, see the "eStream Requirements Document" for information.

Definitions

account

A billing entity consisting of a set of users and subscriptions

user

An entity authorized to use an account

subscription

An agreement between user and the ASP to use an application under terms of licensing.

license

Legal right to use an application at any given time.

account admin

A special kind of user who can add/delete other users from an account.

server admin

Administrator for all the eStream servers and database.

AppID

A unique representation of an application. There is one to one mapping of AppIDs to apps.

FileID

Within an application, a unique representation of a particular file.

access token

This represents the right to run an eStreamed application. The client must acquire an access token before accessing any file (e.g., executing) in an eStreamed app.

application

An *application* is the set of all files and directories, served by an eStream server, that make up a subscribed application. For example, any executable file, DLL, icon file, or data file associated with an eStreamed version of FrameMaker is part of this application.

application installation

This is the process of locally installing all bits necessary to execute an application via eStream. Most files in the application can be read or executed via the eStream file system; some must be installed locally. Some configuration data must also be downloaded and processed to allow seamless execution of apps.

app install block

This is what needs to be downloaded and installed during application installation. It might consist of:

- all configuration files that must be installed on the client machine
- all registry spoofing information required to run the app
- all file spoofing information required to run the app
- the names of all files and directories that make up the application
- initial prefetch data
- initial pages for critical application files

It's quite possible that this app install block is actually an executable file or a DLL that performs all actions to make an application ready to run, rather than simply a block of data.

ASP ID block

An *ASP ID block* consists of all the information about the applications available to a given user for a given ASP, on a given client machine. Since a user might belong to multiple accounts for an ASP, this represents all subscribed applications for all accounts for that user.

Such data might consist of:

- user name
- password
- ASP contact info (IP address, URL, etc.)
- list of subscribed apps
    - is the app installed on this machine?
    - serial number for app
    - ADRM server(s) to use for validation of this app
    - App server(s) to use to retrieve the app install block

  - App server(s) to use to retrieve app file data
- last time stamp when the ASP was checked for new subscriptions

client certificate

The *client certificate* for a client machine is a digital signature used to identify it to eStream servers. We anticipate this to be used for requests that don't require an access token -- i.e., a valid license. For example, retrieving the app install block, or data for eStream application files that don't require license validation.

client machine

This is a computer on which an eStreamed application executes. It may host multiple registered users, and a single user can install the eStream client on multiple client machines.

eStream client

This is the aggregate of all the software required on a client machine to subscribe to, install, and execute an eStreamed application.

eStream file system

The *eStream file system* (or EFS) is a distributed file system with prefetch and caching functionality. All file data and metadata accessed through the EFS is subject to license validation before being available from a server.

license validation

The act of validating a license means gaining an access token. Generally, before an eStream application can be run on a client machine, the validity of using this application by the current user must be checked. This check is done when certain files associated with the application are accessed; an eStream server is contacted to perform this check and return an access token.

subscription serial number

Each application that a user is associated with a serial number. This identifies *both* the application and the user uniquely, and hence can be checked easily during license validation.

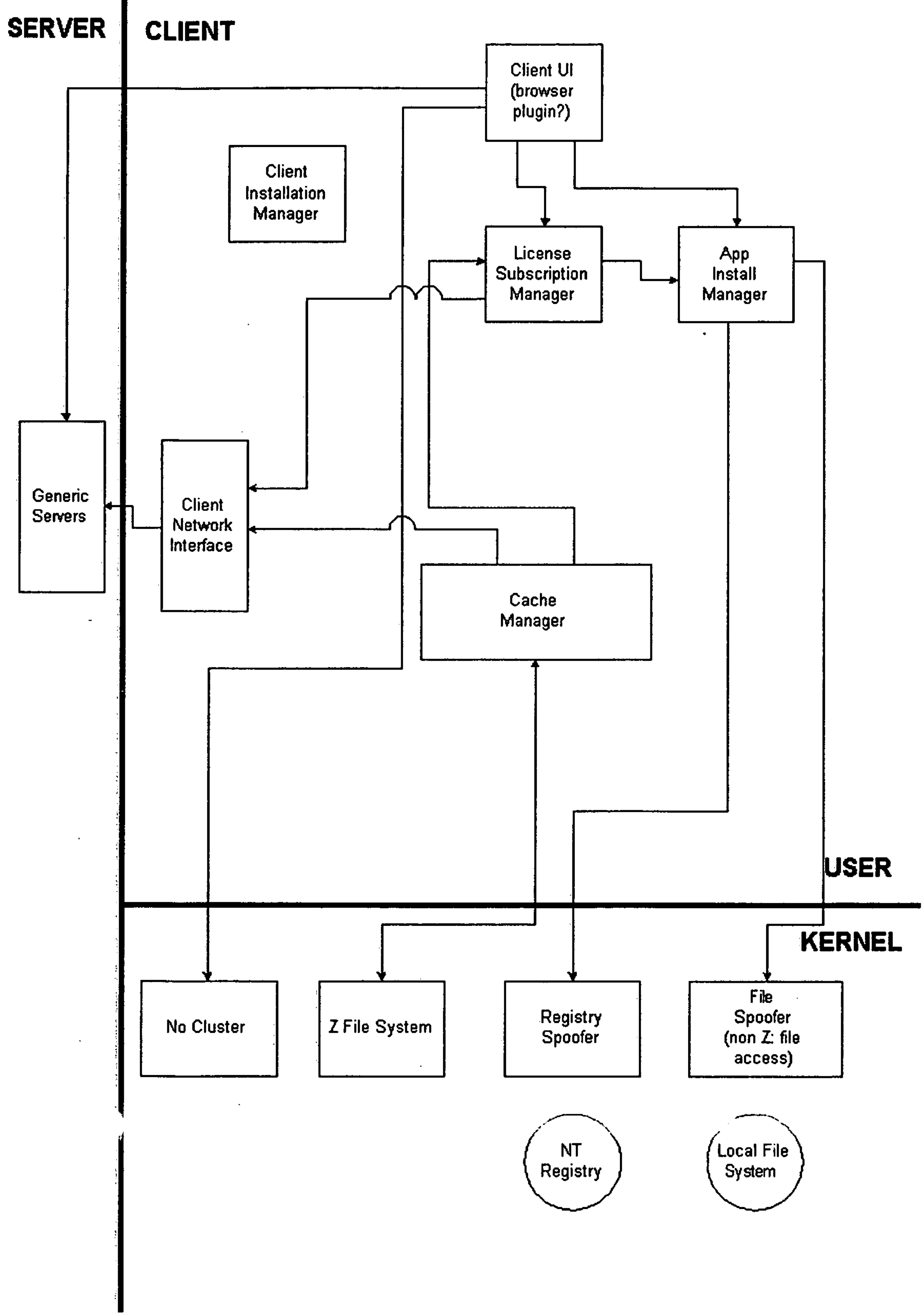
Block diagram

The following are simple block diagrams of the client and server components. Some conventions:

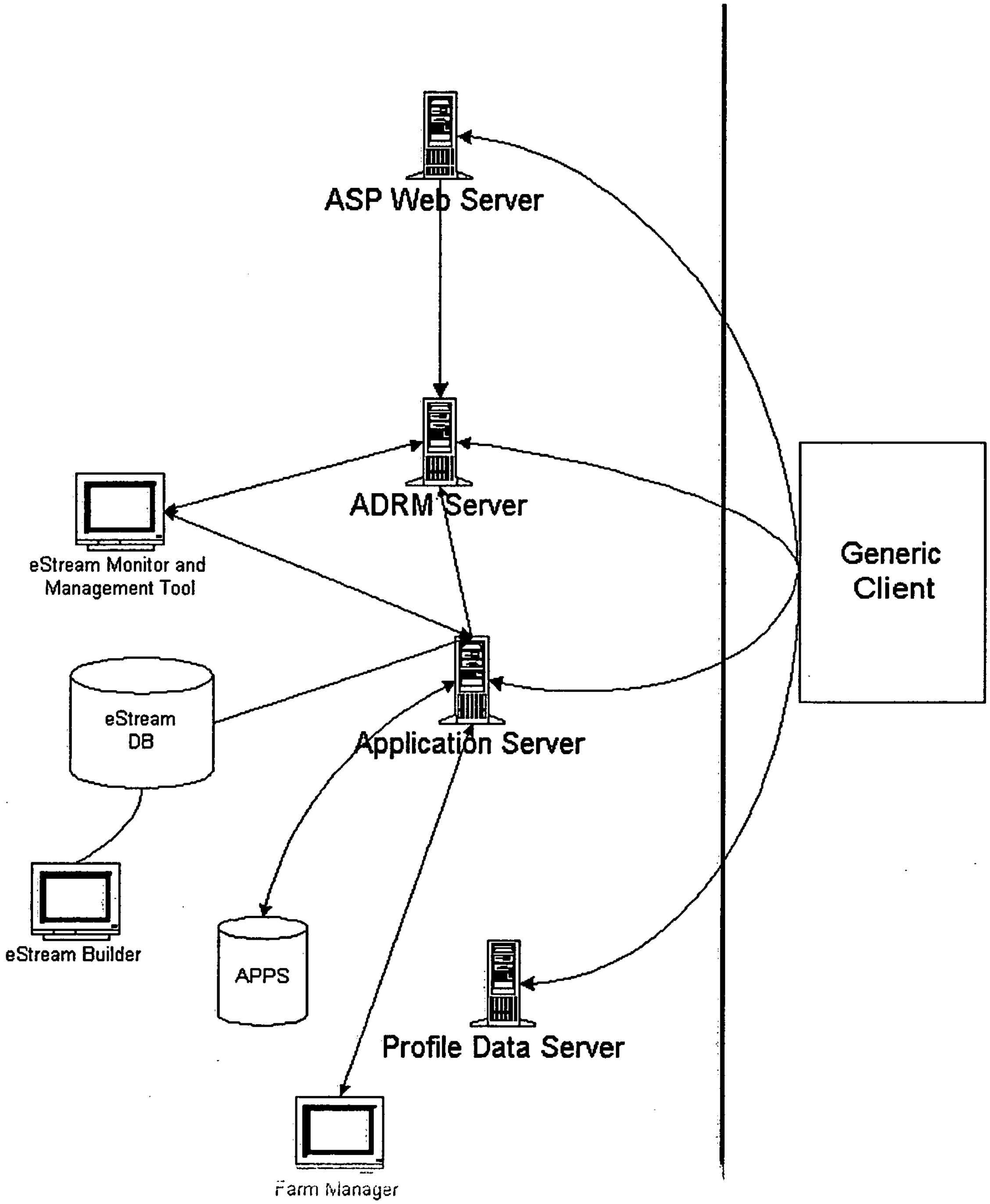
- A box represents a **logical eStream component**. A component may exist as a distinct process, or it may be grouped with other components into a common process.
- A line between components represents an interface call from one to another. If A calls B, there's a arrow on the end of the line at B. If A and B call each other, there's an arrow on both ends of the line.

Note that data stores are **not** represented in these diagrams; if a data store is centrally managed, then there is a component that has interfaces to allow access to these data.

eStream Client Block Diagram



eStream Server Block Diagram



Component descriptions

Client components

The client components are all identified in the block diagram above. Very briefly, some points:

1. A web browser on the client machine will be used for most user interface requests: subscribing to applications, requesting subscription and payment information, and so forth. Configuration of the eStream client software will be done using a UI which may be different from a web browser. Some thoughts on this are listed below.
2. The eStream cache manager is the heart of the client software, and is the component that actually requests file data from the servers.
3. The license subscription manager has the task of tracking all valid subscriptions to applications from an ASP, and tracking which applications have, or need, a license validation to access files.
4. The app install manager's task is to wait until it's told to install a newly subscribed application, and then do so. It also keeps track of what needs to occur when uninstalling an application.
5. The client network interface simply takes requests from the rest of the client components, and forwards them on to the appropriate eStream servers.
6. The eStream file system (EFS, aka the "Z" file system) is a standard kernel-mode network redirector. It presents the normal FS interface to the rest of the NT executive, and requests data from the eStream cache manager to satisfy requests made of it.
7. The registry and file spoofers are kernel-mode drivers that monitor registry calls and file open requests, respectively.
8. The No Cluster component is a very simple kernel-mode driver that disables page clustering for reads.

Installation Manager

Purpose

Installation of the eStream client software is not different than installing any other client software package such as Winzip or Office. The eStream client installation is separate from the installation and configuration of eStream subscribed applications. Some of the possible pieces that eStream would need to be installed are listed below.

1. Device Drivers
2. Applications Executables
3. Application Components
4. Shared Components
5. Registry entries

6. Shortcuts and Start menus
7. Help Files
8. Uninstaller

Once the pieces of the application to be installed are brought together then an install program must be constructed.

Functionality

The Installation Manager consists of the following sub-components

Installshield

Installshield is the industry standard for building installation sets for Microsoft Windows. Installshield will take a set of executables and data files and create a media installation. The Installshield environment provides a scripting language that will allow a high degree of customization of target installation. The essentials issues for any installation are.

1. How much of the application does the user wish to install?
2. Is the users system capable of running the application?
3. Where does the user wish to install the application?
4. Does the user have enough space to install the applications?

Installshield has a wizard that will set up a project. When the install shield program is compiled a media must be specified. The most common media types are floppy, CD Rom, and Web media builds. For eStream we may have to ask the clients to reboot the machine since we are installing kernel mode components that might need a reboot to take effect.

Install From the Web

This program is another product that is sold by Installshield that will take a complete installation set and create a single executable .exe that can be easily downloaded from a web site.

Uninstaller

Installshield will provide an uninstaller when it builds the install program.

Registry Settings

There are three ways that Installshield can patch the system registry.

1. Run regsvr32.exe on self-registering .dll files. When the uninstaller is run it will use regsvr32.exe /u to un-register the .dll file.
2. Patch the registry statically.

3. Patch the registry based on Installation Options from the install shield script program.

Artwork

The Installshield program for eStream will require a splash screen and possibly one or two other artwork components.

eStream Client UI Module

The eStream Client UI module is a client component, currently expected to be running in user space.

Functionality

The eStream Client UI module supports reporting eStream-specific error & informational messages to the client user and soliciting replies when appropriate. It allows the eStream client user to view and change the list of applications currently installed on the client system and the list of ASP accounts currently known to the client system.

Interfaces

ReportMessageToEStreamClientUser(IN message)

Display specified message in EStream Client UI message window.

QueryEStreamClientUser(IN message, OUT response)

Display specified message in EStream Client UI message window and solicit yes/no response for return to caller.

Installed Applications UI

The Client UI interface allows the user to request that the list of the applications currently installed on the client be displayed. The Client UI Module gets this list by calling AIM/GetAppInstallList().

The Client UI interface allows the user to select an application from this list to be uninstalled. The Client UI Module calls AIM/UninstallApp() to accomplish this.

The Client UI interface allows the user to enter the information necessary to get a new application installed. The Client UI Module calls AIM/InstallApp() to accomplish this.

The Client UI interface allows the user to request that the list of applications currently installed on this client be exported to a file, in a form which would allow that list to be

imported on another client. The Client UI Module calls AIM/ExportInstalledApps() to accomplish this.

The Client UI interface allows the user to request that a list of applications that was exported by eStream running on another client be imported from a file & installed. The Client UI Module calls AIM/ImportAndInstallApps() to accomplish this.

Known ASPs UI

The Client UI interface allows the user to request that the list of ASPs currently known to the client be displayed. The Client UI Module gets this list by calling ???.

The Client UI interface allows the user to select and connect to an ASP in the list. The Client UI Module accomplishes this by ???.

The Client UI interface allows the user to select an ASP from this list to be deleted. The Client UI Module calls ??? to accomplish this.

The Client UI interface allows the user to enter the information necessary to record information about a new ASP account. The Client UI Module calls ??? to accomplish this.

The Client UI interface allows the user to request that the list of ASPs currently known to this client be exported to a file, in a form which would allow that list to be imported on another client. The Client UI Module calls ??? to accomplish this.

The Client UI interface allows the user to request that a list of ASPs that was exported by eStream running on another client be imported from a file & installed. The Client UI Module calls ??? to accomplish this.

eStream Cache Manager

Purpose

The eStream Cache Manager (ECM) is a client component, currently expected to be running in user space. Its goal is to:

- Handle all file requests from the eStream file system, either by using previously cached contents or requesting the contents from a server.
- Intelligently use prefetching of file data to reduce latency of pages requested from the EFS.
- Work with the license subscription manager to insure that all applications have appropriately validated licenses before their files are accessed.

Functionality

The ECM handles the volatile & non-volatile eStream cache on the client machine. It performs demand fetching and prefetching from the appropriate server(s), using profiling data or heuristics. Based on the client's observed behavior, it compiles updated profiling data, which may periodically be uploaded to a server.

Interfaces

The ECM takes requests from the EFS driver, and makes requests to the client network and LSM modules.

In the descriptions below practically every call could fail for a variety of reasons. The associated error handling paths are not shown at this level of the design.

Open/Create(IN Filename, IN FileOptions, OUT Handle)

Called from the EFS.

This does the following basic tasks:

- If the filename and mode options correspond to a FileID that is already known and legal to use (from it's cache), it can just return the handle for this file.
- Otherwise, it must ask the LSM for an access token for this file. (This request may simply return an access token previously created for other files making up the application.)
- Launch any prefetching and/or active cache loading activities desired for the new app.
- Request a file handle from an appropriate app server, via the client network component.
- Return the handle to the caller

Close(IN Handle)

Called from the EFS.

This basically:

- Informs the LSM that the file is being closed
- Unloads (or marks as victims) app's cached entries as desired

Read(IN Handle, IN ReadOffset, IN ReadLength, IN BufferPtr, OUT BytesRead)

Called from the EFS.

This does the following:

- Update the profile data for this file
- Check the cache for the requested data
- If not there, request the appropriate pages from an app server, along with any page prefetches that are needed, and place the retrieved data in the cache
- Fill in the buffer and return the number of bytes written to this buffer

Write(IN Handle, IN Buffer, IN Offset, IN Length, OUT BytesWritten)

Called from the EFS.

This will use some copy-on-write scheme. It may be as simple as locking the written structures in the cache.

Global Data

ActiveAppsData Structure

Table of data with an entry for each application that is currently active on the client. Fields for each entry are listed below.

- AppPrefix
- AccessToken
- ServerName
- FilesOpen
- LocalPathName

FileID Type

A globally unique identifier defined for each file associated with an eStream application. All EStream-managed files have these identifiers to allow a common & unambiguous method of file referencing between clients & servers & to simplify switching the client to an alternative server.

ProfileData Structure

Exact contents of this data structure will be defined in the low level design phase; at this point, assume predecessor/successor pairs w/counts.

Volatile & Non-volatile Caching Structures

Exact contents of these data structures will be defined in the low level design phase.

License Subscription Manager (LSM)

Purpose

The LSM tracks current subscription information and determines the need for license validation. It is informed of subscription changes from the client UI, and is queried by the ECM to validate accessibility to different applications, based on the license model for the subscription to that application.

Functionality

The LSM tracks the users subscriptions to different ASPs; it is part of the client component downloaded on a client machine. The LSM starts running when the client component starts running, and is remains active until it stops.

The LSM has a few major tasks:

1. Keep track of what subscriptions the current user has available from all ASPs
2. Determine which application a given file is a part of
3. Acquire an access token to validate a license for file requests that require one

There are two ways that the LSM updates its list of known subscribed applications:

1. It may be informed of new subscriptions, or of applications that are unsubscribed, by the client UI, as part of a browser plugin in conjunction with an ASPs web site.
2. It may asynchronously poll an ASPs ADRM servers to get updated lists of subscribed apps.

When the users start running any of the subscribed eStream applications—i.e., when any eStream'ed file is opened—the ECM queries the LSM before servicing any requests. The LSM checks to see which subscribed application this file belongs to, and, if necessary, gets the appropriate access tokens from ADRM servers along with the identities of application servers that can be used to run the applications; it uses the client certificate obtained when the connection to the ASP was made. At the same time, the LSM can decide to cache the access tokens and the identities of the application servers and decide to serve them directly from its cache.

The ECM informs the LSM when files open and close, and determines from this when applications start and end. The LSM keeps track of when access tokens are expiring and can request for additional access tokens when applications are running and the current one is about to expire.

Global Data

The global data managed by the LSM includes

1. The ASP ID Blocks which are obtained when the user on the machine establishes a connection with an ASP from which the user has subscribed applications.
2. The access tokens and the identities of the applications servers that are obtained from the ADRM servers when the user tries to run the applications.

Interfaces

The LSM exposes the following set of APIs to the client UI:

SubscribeApp(IN ASPId, IN AppID, IN LicenseInfo)

This routine in turn will call the App Install Mgr to install the application on the client machine. This will return a Boolean stating success or failure.

UnsubscribeApp(IN ASPId, IN AppID)

This routine will NOT implicitly uninstall the application. Applications must be explicitly uninstalled. This will return a Boolean stating success or failure.

GetAppList(OUT SubscribedAppList)

This routine will return a pointer to a list of subscribed applications on the client machine.

The LSM exposes the following set of APIs to the ECM:

CheckAccess(IN Path, OUT Root)

The LSM establishes a correlation between the Path and the AppID by querying the App Install Mgr. This routine in turn may contact the ADRM server for appropriate access tokens. This will return a Boolean stating success or failure. At the same time Root will get set to the head of the path that identifies the application so that the file system can use the same access token for everything under "Root".

BeginApp(IN AppID)

To indicate the start of an application. **Note:** this may happen implicitly during CheckAccess().

EndApp(IN AppID)

To indicate the end of the application. **Note:** this may happen implicitly during CheckAccess().

The LSM makes the following API calls.

1. InstallApp(ASPId, AppID) to the App Install Mgr to install the subscribed applications.
2. GetAppId(Path, &Root) to the App Install Mgr to get the AppId from the Path. "Root" is explained above.

The LSM sends messages to the ADRM server for getting access tokens. When a user goes to a new machine and installs the eStream client, the LSM obtains the subscription information from this server when the user first establishes a connection with it.

Application Install Manager (AIM)

Purpose

The AIM is the contact point for installation and uninstallation of applications on a client machine. It gets the requests from the LSM to install applications when the user subscribes to applications, and it gets requests from the Client UI to uninstall applications.

Functionality

The AIM manages application installs on the client machine. It keeps track of what applications have been installed on the client machines, where they have been installed and the various components that are part of the installation. It contacts the application servers (obtained from the ASP ID block) to get the AppInstallBlock. This may be a data block, an application or a dll. The AIM uses the AppInstallBlock to then make the appropriate calls to the Registry and File spoofers; to install some files on the local disk; to "warm" the cache and to update the start menu and other short cuts as needed.

Global Data

The Global Data managed by the AIM includes –

1. The AppInstallBlock obtained from the app server that is used to do the installation.
2. The AppID->Path co-relation that is required to check for access privileges.

Interfaces

The AIM exposes the following interfaces –

InstallApp(IN ASPId, IN AppID)

To install the application using a specific ASP server to get the AppInstallBlock.

UninstallApp(IN AppID)

To uninstall the application from the client machine.

GetAppId(IN Path, OUT Root)

To return the AppID given the Path that is being used to open a file/directory on the eStream file system.

GetAppInstallList(OUT InstalledAppList)

To get a list of the applications currently installed.

The AIM makes calls to the registry and the file spoofers using the AddRegSpoofEntry, AddFileSpoofEntry, etc. APIs.

eStream client network component

This section deals with the components that communicate with the servers.

Purpose

The client network component is the common point of connection between the rest of the eStream client components and the various eStream servers. Any client module that calls an interface of a server does so through the network component.

This component is basically stupid. It knows the protocols needed for communicating with the various servers, and it can encode the requested messages via these protocols, but it doesn't try to be smart with regard to failover, or authentication rejection, or other error conditions. The network component lets its caller deal with such matters.

One design assumption here is that data is received from an eStream server only in response to a request it has made of this server. In other words, all requests originate with the client, never from the server.

Functionality

The client network component communicates with the following servers for the types of requests listed.

ADRM server

1. Validate a user for this ASP and get subscription information
2. Validate a license for a subscribed app

App server

1. Open a file/directory for a subscribed app

2. Various file requests on a previously opened file/directory

Global Data

Probably none (?).

Interfaces

ValidateUser(IN ADRMServer, IN AspAndUserData, OUT SubscriptionInfo)

This interface is called by the LSM; it is used both to validate a user and get updated subscription information for a given ASP.

ValidateLicense(IN ADRMServer, IN APPId, IN ClientCertificate, OUT AccessToken, OUT AppServerList)

This is called by the LSM, to get an access token for an application before its file can be accessed.

AppOpenFile(IN AppServer, IN AccessToken, IN FileDesignator, OUT Handle)

This is called by the ECM, to for any eStream file. Note that this is also used to retrieve an AppInstallBlock, when requested via the AIM. **Note:** the FileDesignator is still undergoing design.

AppReadFile(IN AppServer, IN AccessToken, IN Handle, IN OUT Buffer, IN Offset, IN Length, OUT BytesRead)

This is called by the ECM.

UploadAppProfileDataRequest(IN ADRMServer, IN ProfileData, OUT Success)

It's unclear who calls this!

eStream File System Driver

The file system driver interfaces with the operating system's installable file system facilities, forwards file system requests that it cannot directly satisfy to the ECM, and uses the NT File Cache to optimize repeated accesses to the same data. This component will be very operating system specific, while the interfaces it exposes to the cache manager will be (mostly) OS independent. The file system driver resides in kernel space and implements a portion of the entire eStream file system. Other components, such as the cache manager, the client network interface, and the app servers, implement the rest of the eStream file system. These other components are not necessarily kernel-mode resident.

Functionality

The eStream file system driver (EFS) will send most requests from the operating system to the cache manager. It will interface with the standard NT File Cache Manager to avoid sending redundant requests to the cache manager. And it must support functionality for the ECM to notify it when the data structures it has cached have become invalid.

Global Data

The only globally-visible data managed by the file system driver are various things that it may cache. This includes both file data pages as well as directory contents. These data are relevant to the ECM, because it may want to invalidate the contents of the caches if it finds a newer version of a data page or finds that (visible) directory contents have changed.

Interfaces

These are the logical interfaces that are exposed to the ECM. The EFS has standard file system interfaces that are used by the NT Executive, but these are not listed here.

InvalidatePage(IN FileHandle, IN PageOffset)

Invalidates the specified page for the specified file handle in the cache.

InvalidateDirectory(IN DirHandle)

Invalidates the specified directory's contents in the cache. This may result in the eStream file system driver sending directory change notifications.

ShutdownFileSystem(IN Force)

Attempts to shut down the file system. If Force is true, the file system will be shut down regardless of whether any processes still have handles that are open on the file system. If Force is false, this routine will return an error if there are any open file handles. After the file system is shut down, any attempt to access the file system will result in errors rather than being forwarded on to the cache manager, until StartFileSystem is called.

StartFileSystem()

Causes the eStream file system to begin accepting requests and forwarding them to the eStream cache manager.

Virtual Memory Clustering Disabling Driver

Purpose

The VM clustering disabling driver (aka NoCluster) disables virtual memory clustering under Windows. While we don't fully understand all the implications, using this driver substantially reduces the average file system paging request size and can dramatically improve performance of eStream, especially on slower connections.

Virtual memory clustering, as implemented in Windows NT/2000, is intended to improve performance when paging to and from physical disks. If possible, we would like to disable clustering only for those threads/processes that will be doing a significant amount of I/O to the eStream file system.

Functionality

The VM clustering disabling driver maintains a set of criteria for threads whose clustering should be disabled. It will make decisions about which threads should have clustering disabled without contacting any other components.

Global Data

The only data managed by this component are the criteria for selecting threads whose clustering should be disabled. In the simplest implementation, this would be nothing, and the driver would disable clustering for all threads. Whatever tables it uses will be resident in the kernel, and the driver will be able to access them as needed without making calls to a user-mode component to read them.

Interfaces

The interfaces for this component are minimal. Clustering may be enabled or disabled, and the criteria for which threads to manipulate can be changed.

StartDisablingClustering()

This interface notifies the driver that it should begin disable virtual memory clustering, using the currently specified criteria.

StopDisablingClustering()

This interface notifies the driver that it should stop disabling clustering. Note that due to implementation problems, we do not support actual unloading of the driver, though it can be removed from the system by a reboot.

ChangeClusteringCriteria(IN Criteria)

This interface allows the caller to cause the driver to change the criteria it uses to select threads it should manipulate. The criteria might be specified in the interface, or it might specify a file that the driver should read for the new criteria.

QueryClusteringCriteria(OUT Criteria, OUT Active)

This interface allows the caller to find out what criteria the VM clustering disabling driver is currently using, and whether or not it is currently active.

File Spoofer

Purpose

The purpose of the file spoofer is to redirect file system accesses from some non-eStream drive. This may be necessary in order to support applications running under eStream that are hard-wired to access files in a specific location. The file spoofer may also be used if we are interested in providing a version of some system file different from the one actually on the client machine.

Functionality

The file spoofer will intercept File Create calls for files that we are interested in spoofing and ensure that these creates are redirected to a file we specify. The redirection could be to a file on the Z file system, or to another, non-eStream'ed file.

File open is a very common occurrence, so the file spoofer must operate quickly. The file spoofer should maintain in-kernel whatever data structures it needs to make a spoofing decision.

Global Data

The file spoofer must maintain a database indicating which files to spoor, which file to replace them with, and possibly which processes should be spoofed. All of this information must be kept in-kernel so spoofing decisions can be made quickly. This database may change depending on which eStream apps are currently installed or running.

Interfaces

StartFileSpoofing()

Causes the file spoofer to begin file spoofing, using the current spoof database.

StopFileSpoofing()

Causes the file spoofer to stop spoofing, but does not change the spoof database.

AddFileSpoofEntry(IN SpoofEntry)

Adds an entry to the spoof database. It is not necessary to stop and restart the spoof database to add an entry.

RemoveFileSpoofEntry(IN SpoofEntry)

Removes an entry from the spoof database. It is not necessary to stop and restart the spoof database to remove an entry.

QueryFileSpoofDatabase(OUT SpoofEntryList)

Queries the current contents of the spoof database.

ReplaceFileSpoofDatabase(IN SpoofDB)

Replaces the entire spoof database. It is not necessary to stop and restart the spoof database to perform this action, and it is considered atomic.

1.4 Registry Spoofer

Purpose

The purpose of the registry spoofer is to provide to eStreamed and other applications registry entries for eStreamed apps, and to capture registry writes by eStreamed apps so they can be purged from the registry or shipped to other clients for configuration ubiquity.

Functionality

The registry spoofer must redirect registry reads and writes. Because registry accesses are quite common, the redirector should be able to service registry spoofs without forwarding the requests to a user-level process.

Global Data

The registry spoofer maintains a spoof database of which registry entries to spoof, and which processes to spoof them for. This database should be kept in-kernel so that the spoof decisions can be made quickly. The spoof database may change depending on what eStream applications are currently installed or running.

Interfaces

StartRegSpoofing()

Causes the registry spoofer to begin spoofing using the current database.

StopRegSpoofing()

Causes the registry spoofer to stop spoofing. This does not change the registry spoof database.

AddRegSpoofEntry(IN SpoofEntry)

Adds an entry to the registry spoofing database. It is not necessary to stop spoofing in order to do this.

RemoveRegSpoofEntry(IN SpoofEntry)

Removes an entry from the registry spoofing database. It is not necessary to stop spoofing in order to do this.

ReplaceRegSpoofDatabase(IN SpoofDB)

Replaces the entire registry spoofing database. It is not necessary to stop spoofing in order to do this, and it is considered an atomic operation.

QueryRegSpoofDatabase(OUT SpoofEntryList)

Queries the contents of the registry spoof database.

Server components

The servers described below are **logical servers**. Note that a single server machine can serve all functions for a small ASP; alternatively, farms of servers can be used to provide the functionality of a single logical server.

NOTE: The distribution of servers and the functionality provided by them are somewhat uncertain to date. In particular, exactly who manages the actual accounting, user, and group data is undecided. The group producing the LLD for the eStream servers need to flesh this out. For now, this document assumes that the ADRM server ultimately manages these data, and supplies interfaces to callers to access these data.

The servers described are:

1. An ADRM server handles user/account/subscription data management, as well as validating licenses.
2. An ASP web server is a front-end for requests to add users, subscribe to applications, and do various user and application level queries. Generally this forwards these requests to an ADRM server.
3. An application server handles requests to open and read eStream files.
4. A profile data server will receive uploaded profile data from a client machine to enable better initial profile and prefetch maps in eStream sets.

ADRM Server

Purpose

The Account/Digital Rights Management (ADRM) server is responsible for:

- Managing data related to users, the groups they belong to, and the applications they are subscribed to
- Validating the licenses for applications executing on clients
- Tracking all outstanding licenses currently in use

Functionality

Client machines sends requests to the ADRM server to add or delete subscriptions, to receive an access token to execute an application, and to manage their account/group/user relationship.

Access tokens have an expiration time, so the client must reacquire them at regular intervals. When an eStreamed application exits, the client informs the ADRM server to release the access token. Any outstanding access token not released or reacquired within the expiration time will be automatically released by the server.

Interfaces

AcquireAccessToken(IN UserInformation, IN AppId, OUT AccessToken, OUT AppServerList)

This is called by the eStream client to gain validate a license before executing an application.

This is used to insure that a user has the right to use a particular app in a subscription from a specific account. The server returns an access token and a list of app servers from which the client can access the application file data. If the user doesn't have a valid license to use the requested application, a failure message is sent to the client. The server writes the start time of this application usage into the database for billing processing.

RenewAccessToken(IN OldAccessToken, OUT NewAccessToken, OUT AppServerList))

Note: This may just be replaced by AcquireAccessToken()

This is called by the eStream client.

The server receives a message from a client to renew its access token before the expiration of the token. The server returns a new access token and a list of app servers. This allows the server to redirect the client to a different app server in case it knows of changes to the list of available servers. Once the token is expired, the ADRM server

writes the end time of this usage information into the database and the client must reacquire the access token before files for this application are available to it.

ReleaseAccessToken(IN AccessToken)

This is called by the eStream client.

The client returns the token to the server when the eStream app terminates so other clients can acquire the token. The server writes the end time of this usage information into the database for billing processing.

AddApplicationServer(IN AppServer, IN ApplicationList)

This is called by an application server.

The ADRM server is informed of the availability of a new application server. The ADRM server adds this new app server to its list of app servers.

RemoveApplicationServer(IN AppServer)

This is called by an application server.

The ADRM server is told of the removal of an app server. It must remove this app server from its list of such servers to prevent any clients from using that server.

AddApplicationServerApplications(IN AppServer, IN ApplicationList)

This is called by an application server.

The ADRM server is informed of the availability of a new application on a given app server. The ADRM server adds this new app to the list of applications that the server has available.

RemoveApplicationServerApplications(IN AppServer, IN ApplicationList)

This is called by an application server.

The ADRM server is told of the removal of an application for an application server.

[illegible](IN TimeCriteria, OUT HistoryData)

This is called by a server UI tool, or possibly by other ADRM servers.

The administrator monitoring, reporting, and management tool UI program can query the ADRM server for load information. The server logs all client requests to acquire access token. This raw information can either be sent directly to the UI program or it can be preprocessed before sending to the UI program.

GetErrors(IN TimeCriteria, IN ErrorType, OUT ErrorList)

This is called by a server UI.

The admin UI program can query the ADRM server for any errors. The errors can be categorized by type of errors or errors that occur between certain time periods. A small sample of the possible ADRM server errors includes: client access token timeout, failure to read user information from the account database, failure to get the license information, failure to write usage information into the database, etc...

GetIllegalAccesses(IN TimeCriteria, IN AccessType, OUT AccessList)

This is called by a server UI.

The admin UI program can query the ADRM server for any illegal accesses. The illegal accesses can be categorized by type and time period. A small sample of the possible ADRM server errors includes: failure attempts to access ADRM server with bad password repeatedly in a small time period, failure attempts to use a particular license, any access attempts from a non-typical IP address ranges for a particular account, etc...

GetAppID(IN AppName, OUT AppID)

This is called by a server UI.

Returns a unique identifier associated a particular application

SetApp(ID IN AppName, IN AppID)

This is called by a server UI.

Stores a unique identifier associated a particular application

Application Server

Purpose

The application server is there to handle read requests for files accessed by eStream clients. Any file accessed on a client through the EFS can have this read request passed to an app server.

Functionality

This will be the hardest working eStream server. It will respond to both synchronous (demand fetching) and asynchronous (prefetching) page requests from many different clients, for many different types of applications and files within those applications.

Interfaces

GetFileInfo(IN AccessToken, IN FileID, OUT FileInfo)

This is called from an eStream client. Given any file within an eStream application, return metadata about it. The access token is provided for validation.

ReadFile(IN AccessToken, IN FileID, IN Length, IN Offset, OUT Buffer, OUT BytesRead)

This interface is called by an eStream client, and will allow the client to access any eStreamed application file and AppInstallBlocks. How the FileID for an AppInstallBlock is achieved is unclear at present.

OpenFile() / GetFileID()

Note: This is a placeholder for an API that may be needed. This depends a lot on the eventual communications between client and server for associating a file pathname with a FileID.

ASP web server

Purpose

This describes, of course, only those interfaces on an ASP web server that relate to handling eStreamed applications.

Logically, the ASP web server is the backend web interface for user requests—e.g., get billing information, subscribe to a new app, or request a list of all possible apps a user can subscribe to. In the current model, the web server doesn't actually handle these requests, but instead passes them on to the appropriate eStream-centric server.

NOTE: The following interfaces are not updated from the previous version! They were written with the assumption that the web server actually manages all the data described above. We need the server team to suggest the changes that should take place here!

Functionality

Interfaces

AddADRMServer()

The ASP Web Server is informed of the availability of a new ADRM server. The ASP Web Server adds this new ADRM Server to its list of online ADRM Servers. Periodically, the ASP Web Server can query the list of ADRM Servers for its load

information. When a new client connects to the ASP Web Server, the client can be informed of the subset of ADRM Servers with the least load.

Callers: ASP Web Server

Input:

- ADRM Server IP
- list of Account Ids that the ADRM Server supports.

Output:

- success or failure

RemoveADRMServer()

The ASP Web Server is told of the removal of an ADRM Server. It must remove the ADRM Server from it's locally cached list of ADRM Servers to prevent any future clients from using that particular ADRM Server.

Caller(s): ASP Web Server

Input:

- ADRM Server

Output:

- success or failure

ValidateSubscribedUser()

Inputs:

- SubscriptionToken

Outputs:

- success/failure

Validate subscribed user is called by the ADRM server to check out a license.

- Find account(accountNumber)
- Find user(username)
- Find license(SubscriptionID)

If a license is found to be available, check it out and return OK else return NO_LICENSE_AVAILABLE.

Add/RemoveAccount()

Input:

- ownerUsername
- ownerPasswd
- billingInfo

Output:

- accountNumber

Creator must supply info for the first user, who is also the account owner.

Add/RemoveSubscribableApp()

Input:

- application
- AppServer locations
- name description

AddAccountUser()

Input:

- account number
- user name
- initial password

Add/Remove/Increment/DecrementFloatingLicense()

Input:

- account number
- subscription
- number available

Add/RemovePerUserLicense()

Input:

- account number
- user
- subscription

CompileAccountUsage()

Input:

- account number

Add up and report all the usage by members of the account.

ClearAccountUsage()

FreeLicense()

Input:

- user
- subscription

Frees a license that had been previously checked-out by a user.

GetUserPriveleges()

Input:

- account number
- username

Check privileges of a logged in user for the purposes of allowing user/subscription management and other account changes

ShowCheckedoutLicenses()

GetAccountInfo()

Input:

- account number
- username

ListPossibleSubscriptions()

Input: none

ListCurrentSubscriptions()

Input:

- accountNumber

CreateSubscription()

Input:

- account number
- license data (depends on license type)
- ApplicationPackage

Output:

- subscriptionID
- ADRM server names.

Builder

Does not talk to any other module. Probably an associated set of tools managed via a script plus manual procedures that create intermediate data files, and finally produce the eStream set.

Farm Manager

Simply takes user commands and input, activating the following functions on the actual Application Server process:

StopServer(boolean graceful)

ConfigureServer(config_parameters)

EnableEstreamSet(appID) - informs the app server that the set is ready to be served

DisableEstreamSet(appID) - opposite of above

[illegible] required for [illegible] sets may extend [illegible] ment of farm-level stuff would be an extension of the above. Synchronizing the availability of apps between the Application Server and ADRM/Account DB must also be handled; at least initially, a human administrator should be able to flips the final switch.

Monitor

The monitor utility is responsible for monitoring the overall health of the system. It is responsible to report server status, server traffic, illegal access etc. It will ping the Application Server and the ADRM server to gather the statistics and display them.

Server data objects

This section needs work! What should be here?

eStream Set

What is an eStream set? It consists of:

- Page prediction map, indicates likelihood a page will be referenced successively after another page. Used to enable accurate prefetching by the client.
- Spoofer info, stuff to initialize register & file spoofer to enable application to run on the client. (ITARD & "Spoofed file mapping list")
- Application content. This includes all files of the application in possibly multiple subtrees (stuff from C:, Z:, Common Files, etc.). These are all then placed under a single application root directory which is "mounted" under the eStream file system under the application's directory ("Microsoft Office" or whatever). This directory structure is then processed to create a special eStream metadata file to represent it, and map file names to FileIDs, one assigned to each file for the app, used by the client. All of the application files can then be placed into a single large file with a FileID index in front to allow the **Application Server** to map requested FileIDs to offsets in the large application content image file.
- The above takes care of everything in the AppInstallBlock except for possible COM dlls that might be necessary (TBD what mechanism generates these).

Account/Subscription Database:

Description:

The Account Subscription database manages all the data required to manage accounts, applications and subscriptions for an applications service provider. It is used to verify users and subscription rights, to log usage, and compile billing for application rental. This document describes the data model for the db, the list of accessor APIs, and finally, a list of scenarios that exercise these APIs.

Data Model

The following is a description of the data model for the Account/Subscription Database.

There are two types of records described.

Static Objects are objects that are stored persistently in the Account/Subscription database. The attributes of each record are divided into two categories:

- Owns: specifies data that is actually contained in and managed by the object. These are the attributes that make the object what it is. For example: an account is not an account unless it has billing information. Each item (except the ASP) must have only one owner, but may have many references.
- References: Attributes that create associations to other first-class objects, making navigation from one item to another simpler.

Transient Objects represent data structures that are created in order to pass information from place to another. For the most part, they provide shorthand identifiers for static objects.

Each attribute listed below has the format:

Type: name – (optional) description

For attributes that don't yet have a defined type, the type is *undefined*

Static Data (Database records)

ASP – The top-level container.

Owns:

- List<Account>: accounts - all of the accounts
- List<Subscription>: subscriptions - all of the available subscriptions
- List<Applications>: applications - all of the applications currently served by Application Servers

References:

- None

Account - Collection of attributes that make up a single account:

Owns:

- Number: accountNumber – number that identifies this account
- List<User>: users – all of the account users
- Undefined: billingInfo – Currently undefined type

- List<License>: licenses - all of the licenses (per-user and floating) that are managed by the account
- List<UsageRecord> usage – list of records that describe usage of all account users

References:

- User: owner - a user who created the account – must be one of the users

User – a single user of an Account

Owns:

- String: username
- String: password
- Undefined: Role – specifies permissions on the account, i.e. owner, administrator vs. regular user
- Undefined: UserInfo – Real name, contact info etc

References:

- List<License>: licenses currently held by user
- List<License>: licenses - licenses to which the user has access. These might be either per-user or floating licenses, or a combination of the two. This list might also incorporate a means of specifying a preference – for example if a floating license and a per-user license are available, use the per-user

Application – single application that has been made available on one or more application servers.

Owns

- String: name
- String: description
- Undefined: AppServer location(s) – A location may be a host name that must be resolved by the appserver farm, or may be an IP address.
- Number: AppID
- A list of FileIDs for this app.

References:

- None

Subscription – An application or group of applications that have been made available for rental by a user.

Owns:

- Number: SubscriptionID
- String: name
- String: description

References:

- User: user
- List<Application>: applications - application(s) associated with this subscription

ApplicationPackage – *application(s) that can be rented*

Owns:

- Undefined pricing

References:

- List<Application> applications

License – base for other licenses – All licenses support the same APIs – check out, check in

Owns:

- None

References:

- Subscription: subscription – subscription for which this license grants rights

FloatingLicense – one of a fixed number of licenses distributed to a list of users on a first-come first-serve basis. Contains all attributes in License plus:

Owns:

- Number: numTotal
- Number: numInUse

References:

- List<User> holders – the current holders of this license (length = numInUse)
- List<User> allowedUsers – list of users allowed to check out this license

***PerUserLicense** – license tied to a particular user – one desktop (machine) at a time. Contains attributes in license plus:*

Owns:

- Boolean: isInUse
- Undefined – desktopID

References:

- User: allowedUser – the (only) user allowed to pull this license.

UsageRecord – Describes a billable use of the application

Owns:

- Undefined: Start time
- Undefined: End time

References:

- Subscription: subscription
- User: user

Transient Data

AccountNumber – integer that uniquely identifies a single account with an application service provider.

UserName – string that uniquely identifies a single user **within** an account. To uniquely specify a user to an ASP, it is necessary to qualify the UserName with the AccountNumber of which he is a member. All users have

UserVerifier – combination of the AccountNumber, UserName, and UserPassword. Uniquely identifies a user within an ASP

- AccountNumber
- UserName
- UserPassword

SubscriptionID – Integer that uniquely identifies a subscribable application or collection of applications within an ASP.

SubscriptionToken – describes a user-subscription to the ADRM server. Identifies the subscription as well as the user trying to access it.

- UserVerifier
- SubscriptionID

AppID – A unique numeric representation of each eStreamed application. For example, word := 1000, excel := 1001, office := 1002 etc. We should be able to represent software packages using AppIDs.

FileID – Within an eStreamed app, each app-file gets a unique numeric ID.

eStream 1.0 Requirements

Version 1.1

[redacted]

1.0 Introduction

This document describes the high level requirements for the eStream 1.0 product. These requirements are given first as lists for the client and server components and then as scenarios.

To facilitate the development of follow-on products, eStream 1.0 does not include attributes that explicitly preclude future support of thin clients or of data ubiquity.

2.0 Client Requirements

The following are performance and functional requirements for the client portion of eStream 1.0:

ID	**Description**	**Priority**
1.0	eStream client software operates on Windows 2000, Windows NT4, & Windows 98 for x86 clients.	10
1.1	eStream client software collects profile information during application execution; this information is used to improve client-based prefetching. The profile data may be uploaded [unless user disables this feature] to a server.	8
1.2	eStream client software supports eStream execution of top-selling (as represented by Ziff-Davis suites) desktop & laptop applications; these applications are listed in section 5.0 below. Other applications are supported (opportunistically) as well.	10
1.3	eStream client software is obtained via the web or via some distribution media & is installed via some industry-standard [e.g., installshield] mechanism; its installation requires administrative privileges. [illegible] reboot should be avoided, unless needed to minimize potential stability/reliability problems. eStream client software can be upgraded w/o reinstallation and w/o breaking installed apps.	10
1.4	eStream client software operates across the different languages supported by Windows.	8
1.5	Applications running under eStream have an ave	8

	interactive response time within 10% of client native for connections at 256K bps or higher.	
1.6	eStream client software is able to operate with only 16M of available disk space; this is the minimum supported configuration. User is encouraged to use cache sized for best performance.	8
1.7	eStream client software supports simultaneous execution of multiple eStreamed applications, including multiple instances of a single application.	10
1.8	eStream client software is able to unambiguously reference a particular license for an application.	10
1.9	Applications being eStreamed function in the same way that they would if they were installed locally.	10
1.10	eStream client software tolerates server failure [i.e., it continues running any active apps and allowing apps to be launched], though possibly with some delay, assuming that an alternative server of the needed type is accessible.	10
1.11	eStream client software detects and tolerates lost or garbled messages.	10
1.12	It is difficult to steal an eStream application's code or data from the client.	10
1.13	When an eStream application is being installed on a client, the process detects if the app is already installed & requests user confirmation to continue; a single version of an application is available to the user at a time. Install reboot should be avoided, unless needed to minimize potential stability/reliability problems.	10
1.14	Upon uninstalling an application, all application-specific changes to the client system are removed or undone.	9~~10~~
1.15	eStream client software makes minimal changes to the client system when running, avoiding/hiding registry, DLL, & non-Z file system changes as needed.	10
1.16	eStream client software makes a run/no run license decision quickly enough when an eStreamed application is started not to cause customer satisfaction issues.	10
1.17	eStream client software is launched/terminated at boot/shutdown, at logon/logoff, or on demand.	10
1.18	User is able to set initial size of client cache & to increase the size of the cache later without significant performance penalty.	9
1.19	eStream client software does not include explicit	9

	ASP logon/logoff to run installed apps; ASPIDBlock stored on client machine gets AccessToken to execute app in seamless manner.	
1.20	eStream client software facilitates roaming (i.e., running one's eStream apps from a different client or moving one's client system to a different site).	10
1.21	eStream allows exporting information concerning installed apps on one client to one or more files which can be copied to another client. eStream server info is not invalid/inappropriate when the client is moved to a different venue.	8
1.22	eStream will not allow users to run the same application from multiple clients simultaneously if the license prohibits it.	10

3.0 Server Requirements

The following are performance and functional requirements for the server portion of eStream 1.0:

ID	**Description**	**Priority**
1.0	eStream provides user and account management capabilities.	10
1.1	User Account Creation/Deletion supported.	10
1.2	User Account is able to subscribe/unsubscribe to Applications.	10
1.3	User is able to view billing and account information.	10
1.4	User is able to change password/address/billing information online.	10
1.5	User is able to list all available & subscribed applications.	10
1.6	User is able to access online help/doc, including an FAQ database.	10
1.7	Omnishift provides interfaces to facilitate customer support by third parties.	10
1.8	User is able to enter/modify data securely.	10
1.9	Both IE 4.0/later and Netscape Navigator 4.0/later browsers are supported.	9
1.10	ASP agent [i.e., special administrative user at the ASP] is able to access all user information.	10
1.11	ASP agent is able to disable a user.	10

1.12	ASP agent is able to modify license information for a user.	10
1.13	User is able to add additional users to the account.	10
1.14	Web Server is highly scalable.	10
1.15	Servers are able to operate in non-English language.	9
1.16	ASP may operate eStream system with single server.	9
1.17	Flexible access/export of billing information is supported, to facilitate 3rd party billing systems.	10
1.18	eStream server software and eStream apps can be upgraded w/o impacting installed eStream client software. All upgrades are backwards compatible.	10
2.0	The eStream framework [ASLM Server] provides a mechanism to validate the usage of application components with respect to billing models.	10
2.1	ASLM ~~DRM~~ server is able to validate users to use specific applications.	10
2.2	ASLM~~DRM~~ server records all usage activity down to the granularity necessary to support billing models. The granularity will be reasonably large.	10
2.3	ASLM~~DRM~~ is able to release license on explicit request or timeout from the client.	10
2.4	ASLM~~DRM~~ is portable across a wide variety of platforms and operating systems, including but not limited to: Windows NT4, Windows 2000, Solaris UltraSPARC, and Linux.	10
2.5	ASLM~~DRM~~ servers are fault-tolerant.	10
2.6	ASLM~~DRM~~ servers are scalable.	10
2.7	ASLM~~DRM~~ server is able to report Denial of Service attempts.	10
2.8	ASLM~~DRM~~ server reports illegal accesses.	10
2.9	ASLM~~DRM~~ is able to register its presence/load to the Web Server(s).	9
3.0	eStream Framework provides management and monitoring tool (EMMT) to manage the servers.	10
3.1	EMMT is able to start/stop servers in the eStream framework.	10
3.2	EMMT is able to monitor server activity for all [illegible]	10
3.3	EMMT is able to configure the servers.	10
3.4	EMMT is able to provide historical reporting.	9
3.5	EMMT is able to display information graphically and in spreadsheet format.	8
3.6	EMMT is able to raise alarms on predefined events.	9
4.0	eStream framework provides a mechanism to deploy the application via the eStream Builder.	10

5.0	eStream framework support a variety of licensing models.	10
5.1	Floating license model is supported. n User – k Licenses	10
5.2	Names User License model. (Special case n=k)	10
5.3	Time based licenses at billing granularity.	10
5.4	High water mark license.	10
5.5	Node locked licenses.	8
6.0	App Server is able to Authenticate client's accesses (via AccessTokens) completely locally.	10
6.1	App Server encrypts returned data (via a random key chosen by the client); it must be computationally infeasible to steal an application's code while it is being distributed or to determine which application a client is running.	10
6.2	App Server is as stateless as possible to allow client to switch to alternative app server w/o significant overhead. "Stateless" means that there is no server context that would be lost if the server went down; one classic example of this is that "file open" is recorded on the client, not on the server.	10~~8~~
6.3	App Server is optimized to respond to requests with minimal server load, thereby maximizing scalability.	10
6.4	App Servers may be grouped along with any number of other such servers into a farm with minimal inter-server interactions (as to maximize scalability), with load balancing support.	9
6.5	App Server communicates with clients thru firewalls.	10
6.6	App Server communicates with clients efficiently (e.g., via persistent HTTP connections).	10
6.7	App Server is able to install new eStream sets w/o having to go down.	10
6.8	App Server is robust, able to run for long periods without crashes (i.e. no resource leaks, and handles most/all failure modes for system operations); 24/7 operation.	10
7.0	App Server, ASLM Server, ASP Web Server and EMMT communicate through a database which will include but need not be limited to Microsoft SQLServer.	10

4.0 Builder Requirements

The following are performance and functional requirements for the builder portion of eStream 1.0:

ID	Description	Priority
1.0	The Builder installation monitor runs in the background, when an eStream application is installed as part of its preparation or building capabilities.	10
1.1	The Builder installation monitor captures all the updates to the System Registry that take place during the install.	10
1.2	The Builder installation monitor records all the files created in the two kinds of directories: the install directory and the common directories.	10
1.3	The Builder must be able to gather initial set of application profile data. This data consists at least of the page access pattern for starting and immediately shutting down an application	10
1.4	The Builder must package the eStream Set into an easily manageable packages suitable for ASP administrators to download to their servers.	10
1.5	The Builder must be able to collect per-user profile data from the Profile Server and merge the profile data into a combined data usable for updating the profile data in the appInstallBlock.	8
1.6	The Builder should be run in an environment where no other applications are running.	10
1.7	The Builder should provide functionality to create installation set(s) for each of the clients eStream 1.0 is going to support.	10
1.8	It should be possible to change the appId of the eStream set when an ASP wants to "install" the eStream set in order to host it.	10
1.9	It should be possible to create a merged eStream set for a suite of applications.	10
1.10	It should be possible to test the eStream set created by the Builder using a stand-alone tester and not require the eStream client+server programs.	10
1.11	The appInstallBlock should have support for indicating upgrades at the support site	10
1.12	In the process of creating an eStream set it should be possible for the user to delete file entries and	10

	registry entries manually to "trim" the eStream set if she so desires assuming the user knows what she is doing.	
1.13	The Builder should be run in a clean machine with as few software installed/upgraded as possible.	10
1.14	The Builder should support individual applications in a suite even if the installer of the suite doesn't allow installation of individual applications.	10
1.15	The Builder must be able to create an initial set of cache contents for the eStream client and allow the initial size to be selectable by the user or automatically.	10

5.0 Client Use Cases

5.1 USE CASE: Installation of eStream client code

- Obtain eStream client code bits.
- Install z: file system hooks & setup to have z: mounted at appropriate time.
- Install eStream client code, which services z: file sys requests from local cache or from servers & which handles sideband communication w/ servers, and setup to activate estream client code at time desired by user (boot, login, on demand).
- Install NoCluster.sys to disable page fault clustering at system boot.

5.2 USE CASE: Installation of application

- Obtain AppID & App Server name for installation from ~~DRM~~SLM Server.
- Download AppInstallBlock information.
- Perform initial installation & setup for app, after checking system for previously installed version of app & issuing any appropriate warnings.

5.3 USE CASE: Uninstallation of application

- Remove all registry/DLL/filesys changes associated with app installation.
- Remove all other data associated with application.

5.4 USE CASE: Uninstallation of eStream client code

- Remove z: file system hooks, eStream client code, & nocluster.sys.

5.5 USE CASE: Execution of eStream client code

- Respond to z: file sys requests and detect when new eStream app is referenced.
- Support Client UI requests.

5.6 USE CASE: Execution of application

- Obtain Access Token & list of App Servers from ~~DRM~~SLM Server.
- Contact App Server(s) as desired to obtain file system data.
- Respond to running application's requests, collect usage data. Cache portions of application, file system info, & user preference info.
- Detect server connection issues (apparent loss of connection or connection response below acceptable threshold) & licensing issues; negotiate with A~~DRM~~SLM Server as needed.

6.0 Server Use Cases

6.1 USE CASE: Create an Account

- Customer brings up browser and connects to ASP Web Server
- Screen display shows "create account", customer selects and enters required account info (billing info, owner userid, pword, etc)
- ASP Web server writes account info to Acct DB using **AddAccount** where a unique Account ID is assigned
- Account ID is returned via web page

6.2 USE CASE: Create a User

- Customer brings up browser and connects to ASP Web Server
- Customer enters their userid and pword
- ASP Web server contacts Acct DB, using **AddUser** userid and initial password, gets Acct info and displays to Customer

- Customer selects "add user" and enters required user info (username, address, email etc)
- ASP Web server writes user info to Acct DB updating account info

6.3 USE CASE: Modify Account

(includes disabling an account or user, removing users from accounts, changing pwords etc)

- Customer brings up browser and connects to ASP Web Server
- Customer enters their userid and pword
- ASP Web server contacts Acct DB, passes along userid and pword, gets Acct info and displays to Customer
- Customer selects "update info" and enters desired changes
- ASP Web server writes updated account info to Acct DB

6.4 USE CASE: AddSubscription

- Connect to ASP web server
- Enter account number, username, password
- Verify that user is account admin using **GetUserPermissions**
- Get list of possible subscriptions (using **ListPossibleSubscriptions)**
- Get list of current subscriptions for account (using **ListCurrentSubscriptions)**
- Display in page – User chooses a subscription and license type
- Display a screen to allow the user to configure the license. For a floating license, allow selection of users, etc.
- Call **CreateSubscription** to compose the new subscription for each user and create licenses.

6.5 USE CASE: Building an eStream set:

- Start w/app **CD-ROM**, and a freshly installed OS (plus latest service pack?).
- Install app into Z: drive (could just be a regular network drive)
- A **special system monitor** logs all registry changes and file system changes during the install.
- File system changes to C: during install probably need to be spoofed (or have a registry entry point to Z: instead), especially newly added directories, so need to do the appropriate thing.

- From this log and the actual files as installed on the machine, the **eStream set builder** creates the eStream set, which is a small set of related files.
- Separately, we need to actually set up the app for eStreaming, then run it and collect profile data to seed the initial page prediction map.

The **AppServer UI** (interface to user to control an Application Server on a particular machine) presents the following management functions:

A. Starting a server:

- **AppServer UI** always indicates whether an AppServer process is up and running (and alive w/status), and if present prompts for restarting the current server process.
- Otherwise it goes ahead and starts up the AppServer process and reports any errors.

B. Stopping a server:

- Simple, just stops any running servers, gracefully, perhaps prompting user for ungraceful shutdown if not successful.

C. Install eStream set:

- Each server is configured with a specific eStream set directory, under which it places (in their own individual directories) the actual eStream set contents (a few files on the native file system).
- User indicates to **AppServer UI** where to find the eStream set package provided by Omnishift. **AppServer UI** authenticates the package, and verifies its integrity, and if successful, unpacks and places the constituent files in the server's eStream set directory.
- Note that it is possible for the eStream set directory to live on a file server shared by other **Application Server** machines, so installation may be required only once (otherwise it must happen once per machine, and a separate **AppFarm Manager** is responsible for replicating eStream sets across a farm to ensure the farm machines are symmetric).
- How does the server know a new eStream set is available? Each set is assigned a VolumeID, and the set contents can be placed under a directory with the same name as the VolumeID. The install is synchronized via a AppList file, which just lists the valid VolumeIDs, which the **Application Server** only reads, so an entry is added at the end of the install procedure. The **AppServer UI** then must send some kind of message/signal to the Application Server to have it resync with the file (and start serving the new app).
- Also note that eStream set install is doable without bringing down the server (or any server in the farm.

- Having done this, it will probably be necessary to notify the ~~DRM~~SLM and **Account Servers** that a new app is available. With some scripts provided by omnishift, this sould be done by a human administrator. They need to know the VolumeID of the app that was installed along with the full name, so that the client can initiate an app install procedure via the VolumeID (the server can then provide the **AppInstallBlock** which probably has a fixed reserved global FileID).
- Questions: What if app is already installed (want to allow reinstall or force remove first)? What if app is being upgraded (probably also should be a remove and then install)?

D. Remove eStream set:

- First we probably have to disable the application on the ~~DRM~~SLM/**Account servers.**
- This probably will require sending some kind of message to the Application Server (if running) to stop serving the given eStream set, and then waiting for any active connections to expire.
- Then we can just remove the entry in the AppList file, and delete the file system image.

E. Configure Application Server:

- The **AppServer UI** presents various configuration options to the user (stuff like logging, port #, threads, etc.) Some may require restarting the **Application Server** to take effect, others may take effect immediately.

Another activity that occurs, automatically, is the processing of profile data. It is not clear what the page prediction map looks like, but clients will periodically send profile data to the **Application Server**, which aggregates it, and must store it persistently, and to allow new clients to benefit from improved prediction. There may need to be a special module that can take the aggregated profile data to modify the prediction map.

6.6 USE CASE: Acquire Access Token

Where are we?

- Client PC has installed the subscribed app and has received a subscription token, and the name/IP of ~~DRM~~SLM Server.
- Customer is accessing an app file and doesn't have an access token for it, yet. (i.e. double clicking z:\word.exe).

Players involved: client – cache mgr, ~~DRM~~SLM Server and indirectly, User/Account/Sub/Rights DB.

What happens:

- Client contacts the ~~DRM~~SLM server and gives: subscription token, user/passwd.
- ~~DRM~~SLM server looks into the user/acc/sub/right DB to
  - Authenticates user and password; may return: "invalid user".
  - Authenticates subscription token; may return: "invalid subscription token"
  - Look at the Accounts container and see if any licenses are available. If so, check it out by creating a new access token and updating the accounts container. It may return: "can't get license".
- Return an access token to the client and a list of app servers.

6.7 USE CASE: Process File Request – steady state

Where are we?

- Client has installed the app and has a list of app servers,
- Client is holding a valid access token that it acquired from the ~~DRM~~SLM server.
- Client, while processing an IRP, needs to access portion of file on the app server.

Players involved: client – cache mgr, app server. NO ~~DRM~~SLM server or no user/account/dub/rights DB.

What happens:

- Client contacts one of the app servers and gives: access token, App ID, File ID, length and file offset.
- App server quickly verifies the expiration date on the access token.
  - It **must not** need to contact the user/account/sub/rights DB to do this. It only cares about the time-validity of the token. If token has expired, return some kind of an error back to the client.
- App server locates the data and sends it back to the client.
  NOTE: we are simplifying this quite a bit when discussing the scenarios because we are not sure exactly how we are going to manage the server farms. Another key question is MUST all app servers host all estream apps ?

6.8 USE CASE: Renew an Access Token – steady state

Where are we?

- Client acquired an access token from the ~~DRM~~SLM server.
- While running the app, client sees the needs to renew the access token. This may happen synchronously when the user touches one of the app files, or by a timer-driven client daemon that periodically renews an access tokens before it expires.

Players involved: client – license manager, ~~DRM~~SLM manager, and indirectly, user/accounts/sub/right/ DB.

What happens:

- Client sends an access token to the ~~DRM~~SLM server.
- Check the time-validity of the access token.
  Assumption: ~~DRM~~SLM server assumes that only valid access tokens can be renewed. An expired token implies a lack of renewal, which implies releasing the license. ~~DRM~~SLM server can try to acquire the license, but there is no guarantee that it will succeed.
    - If token is expired app, goto Scenario: Acquire Access Token.
- ~~DRM~~SLM server accesses the user/account/sub/rights DB to:
    - Generate a new token that will expire some time in the future (configurable parameter).
    - Update the account container in user/account/sub/rights DB.

  ~~o~~Return the new access token.

6.9 USE CASE: Validate user request for access to an application server

Procedure:
Receive username, password, machineNodeID, subID and appID from the Client
Query AccountDB for license to access application appID in subscription subID
If (no valid license) then
 Send FailureReason to Client
Else
 Send accessToken, appServers to Client

6.10 USE CASE: Add subscribable application from an account

Interface Required:
~~DRM~~SLMServer::AddSubscribedApp(accountID, subID)
Procedure:
Receive accountID, and subID from the Client
Check for valid accountID, and subID on AccountDB
If (no valid accountID or subID) then
 Send FailureReason to Client
Else if (subID is not already subscribed under accountID)
 Add Subscription subID to Account accountID in AccountDB
Send Success to Client

6.11 USE CASE: Remove subscribable application from an account

Interface Required:
~~DRM~~SLMServer::RemoveSubscribedApp(accountID, subID)
Procedure:
Receive accountID, and subID from the Client
Check for valid accountID, and subID on AccountDB
If (exist subID in accountID) then
 Remove Subscription subID from Account accountID in AccountDB
 Send Success to Client
Endif
Send FailureReason to Client

6.12 USE CASE: Monitor/management tools

Interface Required:
~~DRM~~SLMServer::GetTrafficHistory()
~~DRM~~SLMServer::GetUsageInfo(userID, appID, subID, accountID)
~~DRM~~SLMServer::GetCurrentTraffic()
~~DRM~~SLMServer::AddServer(serverID)
~~DRM~~SLMServer::RemoveServer(serverID)
~~DRM~~SLMServer::RemoveClient(userID, serverID)
~~DRM~~SLMServer::GetErrors()
~~DRM~~SLMServer::DumpErrors(filename)
~~DRM~~SLMServer::DeleteErrors()
AppServer::GetTrafficHistory()
AppServer::GetCurrentTraffic()
AppServer::GetErrors()

Procedure:
~~DRM~~SLM Servers keep track of traffic info. The monitor/management tool can query the ~~DRM~~SLM/App Servers anywhere for traffic info. Some examples of traffic data:
- Traffic history of particular server on number of clients served per unit time
- Monitor length time a userID used application appID under subscription subID and charged to accountID
- Monitor current load information on all servers (~~DRM~~SLM server and app server)
- Allow admin manually add/remove some servers from the pool.
- Allow admin to kick some clients off the server.

The monitor/management tool can also be used to display a list of errors logged by the servers.
- Monitor errors and be able to categorize by error type
- Monitor errors occurring between certain time periods
- Monitor errors reported by a particular server
- Manage errors to dump the errors to a file
- Manage errors and delete a subset of errors

Finally, the monitor/management tool can check for any illegal accesses.
- Monitor failed attempts to access ~~DRM~~SLM Server with bad password, especially on repeated failed attempts in a short time frame.
- Monitor any attempts to use a particular license and failed.
- Monitor access to ~~DRM~~SLM Server from non-typical IP addresses for a particular account. The server is required to save the history of IP addresses of accesses to a particular subscription account.

6.13 USE CASE: Adding a new application server.

Summary:
An application server's functionality is to provide applications eStream sets to client application. An application server is generally added to the system to provide greater scalability and/or to provide additional application support.

Actors:
1. ASP administrator: Responsible for installing the server and the applications.
2. A~~DRM~~SLM Server(s): The A~~DRM~~SLM server needs to be notified of the presence of an additional application server and the services it provides.

Inputs:
1. Application server(AS) installer
2. Application eStream sets. These may be available from one of the following location: AS installer, some other AS or Farm Manager Server(some central repository) .

3. ~~DRM~~SLM Server location(This input may not be required based on scalability solution that we decide on).

Processing:

1. Using the AS installer install the application server.
    a. <Server install use case to be added here? Later>
2. Copy the Application eStream sets. There are several options here:
    a. Provide the eStream sets as a part of the installer.
    b. Provide a script to ftp to another Application server and copy the eStream sets.
    c. Provide a management tool to manage the copying of the eStream sets. From the ASP's perspective this is the best solution. A tool which provides tracks the application would be useful to manage the load.
3. Configure the server. The server needs to know the additional application sets that it supports(? This may not be required).
4. Start the server.
5. Register the server with other ~~DRM~~SLM servers. The following options apply:
    a. Multi-cast the "new server and services" message to the ~~DRM~~SLM servers.
    b. Register the server to a local object server which in turn notifies the object servers across the system. CORBA model supports this.
    c. Using the resonate model(described below), all appservers are essentially the same server. ie Address app.foo.com will point to a set of app servers. A new server enabled will resonate software will automatically register itself with the resonate scheduler. (How do we make the resonate scheduler aware of the applications available on the app servers?)

Outputs:

1. The App server is installed and running with a set of applications available on it.

6.14 USE CASE: Removing an Application Server.

Summary:
An ASP administrator may decide to remove an application server from the system for various reasons. Removal of server from the system would result in notification to the rest of the ~~DRM~~SLM servers that it is no longer available for servicing the objects.

Actors:

1. ASP administrator.
2. ~~DRM~~SLM Servers.

Inputs:

1. Application server running on the machine.
2. A~~DRM~~SLM Server(s): The A~~DRM~~SLM server needs to be notified of the presence of an additional application server and the services it provides.

Processing:

1. Stop the application server. This will result in the Application server informing the rest of the A~~DRM~~SLM servers that it will no longer take any requests. This in turn may result in an application being unavailable for usage. Depending in the framework used, this can be done in one of the following ways:
    a. Multi-cast the message to the A~~DRM~~SLM servers.
    b. Just stop the server in the CORBA framework. The local ORB server will notify the unavailability of the resource to the rest of the framework.
    c. Using the resonate model to scale would imply that you just stop the server the resonate agent on the server will notify the resonate scheduler to deregister the servers. (However its not clear if you can also deregister the objects served by the server.).

Outputs:

1. A~~DRM~~SLM servers are notified of the removal of the resource.

6.15 USE CASE: Add a new A~~DRM~~SLM server.

Summary:
The ASP provider may decide to add an additional A~~DRM~~SLM server to enhance the performance of the system. The additional A~~DRM~~SLM server added to the system should be accessible to the ASP's Web Server so that it can direct the clients to the ~~DRM~~SLM server. (This may not be required if we deploy the Resonate model of scaling).

Actors:

1. The ASP administrator.
2. The ASP Web server.

Inputs:

1. A~~DRM~~SLM installer
2. Web Server location(This input may not be required based on scalability solution that we decide on).

Processing:

1. Using the A~~DRM~~SLM installer install the application server.
   <Server install use case to be added here? Later>
2. Start the server.
3. Register the server with ASP Web Servers. The following options apply:
    a. Multi-cast the "new server and services" message to the Web servers.
    b. Register the server to a local object server which in turn notifies the object servers across the system. CORBA model supports this.
    c. Using the resonate model(described below), all A~~DRM~~SLM are essentially the same server. ie Address a~~drm~~slm.foo.com will point to a set of A~~DRM~~SLM servers. A new server enabled will resonate software will automatically register itself with the resonate scheduler.

Outputs:
The A~~DRM~~SLM server up and running.

7.0 Builder Use Cases

7.1 USE CASE: Install Monitoring

- Query builder for CD media and installation executable(s)
- Monitor various registry and file updated during installation
- Merge installation data for all applications in a suite
- Relocate files from C: to Z: directory
- Create appInstallBlock and package the appInstallBlock with the application files

7.2 USE CASE: Profiling

- Query builder for application executable(s)
- Monitor sequences of file accesses from OS to the file system as profile data
- Identify the subset of the profile data as the initial cache contents
- Merge profile data and initial cache contents into the corresponding appInstallBlock

8.0 Key Applications for eStream 1.0

Winstone99:
Business:
Corel®WordPerfectSuite8: Quattro®Pro 8, WordPerfect®8, Netscape Navigator®4.04
Lotus® SmartSuite®: Lotus® 1-2-3® 97, Word Pro® 97, Netscape Navigator® 4.04
Microsoft®Office97: Access97, Excel97, PowerPoint®97, Word97, NetscpNav® 4.04
High-end:
Adobe® Photoshop® 4.01, Adobe® Premiere® 4.2, AVS/Express® 3.4, Bentley System's MicroStation® SE, PV-Wave® 6.1, Microsoft® FrontPage® 98, Microsoft® Visual C++® 5.0, and Sonic Foundry® Sound Forge® 4.0.
Content Creation Winstone 2000:
Adobe Photoshop 5.0, Adobe Premiere 5.1, Macromedia Director 7.0, MacromediaDreamweaver2.0, NetscapeNavigator 4.6, Sonic Foundry Sound Forge 4.5

Please note that release of Business Winstone 2000, which was originally slated for [redacted] has now been postponed until the Fall Comdex & will be called Winstone [redacted] As soon as the contents of this suite are released, we should move quickly to assess our support for its application set.

eStream 1.0 Server Scaling Estimate

Anne Holler *[redacted] * *Version 1.0*

Introduction

This document presents an estimate of server scaling for the eStream 1.0 product as compared with its chief competitor, the Citrix product as deployed by Personable. The document presents relevant attributes of the basic application execution model for each of the two products, discusses and gauges the impact of the areas in which server scaling differs between them, considers the effects of additional attributes of the two products on server scaling, & finally summarizes the differences in expected server scaling in terms of a number. Please feel free to challenge the assumptions, methodology, & calculations herein, now & as we move forward through the design & implementation phases.

In the process of developing this server scaling estimate, certain assumptions about user, system, & program behavior are made, & certain design/implementation goals of the eStream 1.0 product are assumed. This material is listed in separate sections at the end of the document for ease of reference.

This work does not intend to imply that the user experience of the eStream 1.0 & Personable/Citrix products is expected to be comparable with respect to the relative server scaling point identified. Interactive response differs between the two products; first-hand experience with server-based applications running on New Moon & Personable/Citrix and client-based applications running on the eStream prototype suggests that the former are sluggish on an ongoing basis with respect to activities such as selecting from pull-down menus & the latter are as responsive as native wrt such activities, with noticeable delays engendered only when heretofore unused portions of the application's functionality are exercised. Reliability also differs between the two products; smooth fail-over of an active Personable/Citrix application to another server is not supported, whereas such fail-over is included in the eStream 1.0 design.

This document does not address an area in addition to server scaling that may be of competitive interest to ASPs; that area is network bandwidth differences between eStream 1.0 & Personable/Citrix. Though it might seem intuitive that sending application pages across a network consumes more bandwidth than sending user input & display output, aggressive client caching ameliorates the traffic associated with application paging, whereas the traffic associated with the display output can be quite substantial, according to Harwood's book "WNT Terminal Server & Citrix Metaframe" [hereafter, HARW99]. It may be worthwhile to collect and compare bandwidth data wrt the two products.

Acknowledgements

Amit Patel provided key insights. Any mistakes/bogosity are mine.

Application Execution Models

The following are basic attributes, with respect to application server scaling, of the application execution models of Personable/Citrix and eStream 1.0:

Personable/Citrix Client/Server Application Execution Model
Application executes on server
On app page faults & app data reads, page loaded from server disk
Server handles incoming network traffic for all keyboard & mouse events
Server handles outgoing network traffic for bitmap display update

eStream 1.0 Client/Server Application Execution Model
Application executes on client
On app page faults & app data reads, page loaded from client cache, excpt miss to server
Server handles incoming network traffic for client cache misses
Server handles outgoing network traffic for client cache misses

Application Server Scaling Comparison

The impact on server scaling of executing applications on the client, rather than on the server, is expected to be large. Processor & main memory overhead are associated with executing applications, including the overhead for fielding interrupts such as mouse & keyboard events. According to HARW99, processing power for running applications is typically the biggest Citrix product bottleneck, and that is reinforced by the variance in Citrix scaling numbers reported wrt Personable [Ernie] and wrt another ASP [Amit], for which the main differences seem to be application execution overhead. HARW99 indicates that Citrix scales at 10 to 45 applications per processor, largely due to execution overhead (though some overhead is due to processing page faults & accessing application data, which is considered in the next paragraph). It is somewhat difficult to understand how to model the relative benefit for this difference (in the sense that an *infinite* number of applications can run on a processor that they are not actually using to execute!); it seems conservative to assume we get at least as much benefit from this factor as we do from reducing cache miss overhead on the server, so let us have this factor double whatever benefit we project from the factor considered in the next paragraph.

The impact on server scaling of processing application page faults & application data reads on the client in most cases, rather than on the server, is expected to be measurable. For the purposes of the server scaling estimate presented in this document, let us assume that the server overhead to fetch a page from disk, whether the request came from server execution or from client request, is comparable. (Although we can construct file server technology in which client requests take less time than server requests, let us assume that the overhead to encrypt the response consumes that time savings). Also, let us assume that the same number of page faults occur on the client & on the server (which server partitioning for Personable/Citrix could render untrue.) Hence, the difference in server

overhead for application file accesses is estimated to be equivalent in scale to the reduction in the number of references, which is derived from the client cache miss rate. Assuming a client cache miss rate of 2%, each eStream application server can handle 50 times as many clients with respect to this attribute of server overhead. Doubling that amount due to the factor described in the previous paragraph, we estimate that each eStream application server can handle 100 times as many clients as each Personable/Citrix application server.

It is somewhat difficult to know how to compare the network interface overhead component of server scaling between eStream 1.0 and Personable/Citrix. The loading constraints associated with executing applications on the server are expected to limit the amount of network overhead presented to a Personable/Citrix application server. Depending on the kind of application, significant traffic is generated to support client displays, but HARW99 identifies network bandwidth overhead [discussed in the Introduction section] – not server overhead - as the scaling problem engendered by this traffic. Each eStream 1.0 client generates little server network overhead, but the reduction in server load due to client application execution allows more clients to be connected to a given server, possibly straining the network-oriented portions of an eStream application server. At this point, let us assume that server network interface overhead does not materially impact the relative server scaling of the two products.

SSL encryption can dramatically decrease server scaling; by more than 70%, according to Igor Balabine. He indicates that third party SSL accelerators should be employed to remove this overhead.

Additional Server Scaling Considerations

For eStream, application installation induces load on the application server to deliver to the client the contents of the AppInstallBlock, which may contain registry & file spoofing information, initial cache & profile data, file system structure information, etc. Personable/Citrix does not have a comparable feature. Installation is expected to be an infrequent occurrence and Omnishift is expected to suggest/provide mechanisms to smooth out (or specially handle) high peak demand at particular points in time, including product or application launch/upgrade. Let us assume that this (managed) overhead reduces application server scaling by the equivalent of 0.5% cache miss. Adjusting the running total by this amount implies that an eStream application server can handle 67 times as many clients as a Personable/Citrix application server can.

For eStream, client prefetching from the server can cause application server overhead that does not have a counterpart on Personable/Citrix [except for page fault clustering ;-) !]. Given that eStream 1.0 is targeted at fat clients, client prefetching (which is redundant wrt a comparably warmed client cache) is expected to be used sparingly. Let us assume that prefetching adds the equivalent of an additional 0.5% cache miss rate; the intuition here is that prefetching is essentially engendered when cache misses occur (i.e., when we are exercising parts of the app we have not exercised before) and that we get unneeded pages a measurable percentage of the time. Updating our running total by this amount

means that an eStream application server can handle 50 times as many clients as a Personable/Citrix application server can.

Both eStream & Personable/Citrix products include several logical servers in addition to the application server. Both have an ASP Web Server portal to an ASP's account services, from which a user can obtain billing information, get a list of available applications, subscribe to new applications, etc. For both, the ASP web server interfaces in some way with an account database of presumably comparable complexity. Both eStream & Personable/Citrix have server functionality involving getting the license to run an application, which is expected to cause similar database overhead; in eStream 1.0, this process involves getting an AccessToken from an ADRM server. However, Personable/Citrix does not have eStream 1.0's concept of renewing an AccessToken. It is expected that the eStream 1.0 design will take special care that renewal does not add significant overhead to the eStream 1.0 ADRM server (by specifying nontrivial billing granularity & AccessToken renewal frequency, by ensuring renewal is lightweight, perhaps by having some explicit mechanism aside from AccessToken renewal for token cancellation in the event of client failure/disconnect, etc.). eStream 1.0 has the concept of records containing application profile information being uploaded to a server; it is not known that Personable/Citrix has any comparable feature (although the product likely has much other user information recorded at the server, including preferences, etc). It is expected that eStream 1.0 design will emphasize minimal server impact for handling this data. [Profile data may not be a core deliverable for competing with Personable/Citrix.] In summary, it is assumed that server scaling for auxiliary servers is comparable between eStream 1.0 & Personable/Citrix.

Conclusion

Based on the discussions in the previous sections, eStream Server Scaling expected to be significantly higher than that of Personable/Citrix. Considering client execution benefits, client caching benefits, application installation overhead, and prefetching overhead, eStream server scaling is expected to be approximately 67 times higher than the Personable/Citrix competitive product. For some given Personable/Citrix application server that can handle 20 clients, an eStream application server can handle 1340.

Assumptions Underlying Server Scaling Estimate

User's application usage pattern [what is run, for how long, what features] does not change materially depending on whether s/he is using eStream 1.0 or Personable/Citrix. It would be interesting to collect data on typical usage patterns; Sridhar formulated estimates for his network bandwidth evaluation, but it might be interesting to see if our in-house use of common applications matches those estimates.

Overall application server capacity is adequate for both products. Personable/Citrix may deny application server access to clients when insufficient overall application server capacity is available, whereas eStream 1.0 may continue to allow clients to access the

servers without some explicit client capacity cutoff point, given the usual small impact of each additional eStream client. However, it is possible that an extreme performance collapse could occur on eStream, if client load were to grow so large compared with the capacity of eStream's application servers that the lack of server response caused an escalating number of redundant requests from eStream clients retrials. This situation needs to be avoided, in the eStream 1.0 design and/or in its deployment.

The increased client servicing capacity afforded an eStream 1.0 application server will not uncover some insurmountable system bottleneck never encountered with the limited client servicing capacity possible on the Personable/Citrix server, including areas such as maximum number of threads, processes, sockets, buffer size for socket send or receive, socket listen queue length, network buffer cache, maximum number of file handles, etc.

eStream 1.0 clients are fat enough to allow adequate client caching to hit or better the client cache miss goal of 2%. Thin clients (which are not the intended design target for eStream 1.0) or fat clients with inadequately sized client caches would increase server overhead for paging requests & network traffic beyond the levels estimated in this document.

Design Goals Supporting Server Scaling Estimate

Overall client cache miss rate is less than 2%. Reaching the eStream 1.0 client performance goals also reinforces the need for a low client cache miss rate. We have not collected data indicating how large a client cache would be needed to hold application pages and file metadata associated with typical application usage as represented by the Ziff-Davis benchmark runs. I think it would be useful to have such data, since it may influence client cache design & effective cache management policies.

AppInstallBlock server overhead is no more than the equivalent of an extra 0.5% cache miss rate. Installation is expected to be relatively rare, & downloaded material is expected to be kept at the minimum size necessary to reach our functionality & client performance goals.

Client wasted prefetch overhead is no more than the equivalent of an extra 0.5% cache miss rate. With fat clients & large warm caches, prefetching is expected to be kept at a minimum for eStream 1.0; again, data gathering related to this area may be useful.

eStream License Manager (LSM) Low Level Design

Charles T. Booher
[redacted]

Functionality

The LSM tracks the users subscriptions to ASP accounts. It is part of the client component and runs entirely in user space. The LSM is part of the client UI program that also contains the Cache Manager (ECM), Application Install Manager (ECM) and client user interface components. The LSM determines when an application has a right to run on a client machine. This is done through the acquisition and management of access tokens. The LSM also provides a server list to the ECM (Cache Manager) for each application that is subscribed. The LSM has two main tasks.

Manage and provide lists of servers for the ECM

Provide access tokens to the ECM and renew them before they expire.

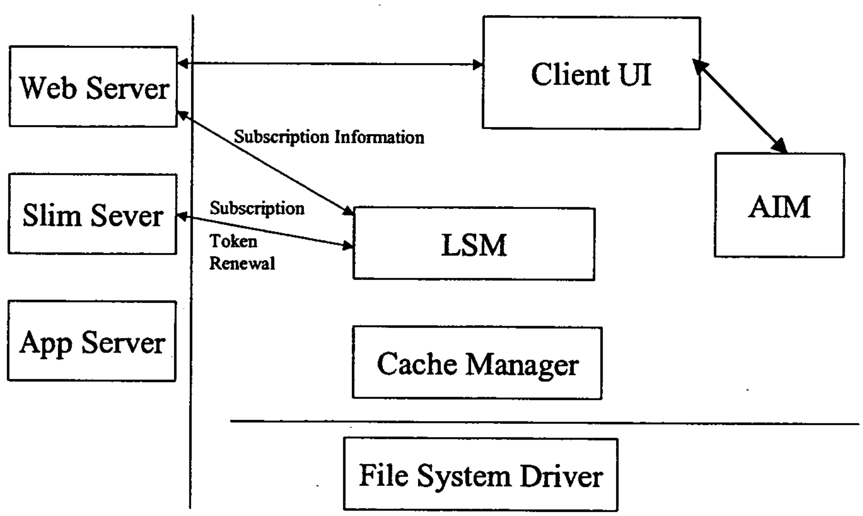


Figure 1 LSM data flow

What does the LSM do?

The LSM manages a set of tables inside of an mdb database. This database can be accessed with ADO or ODBC. This database includes the following tables.

Application Table: This is a list of all the applications subscribed.

User Table: List of User account information

Application Server Table: This is a list of application servers for each application subscribed by the client.

Slim Server Table: This is list of all Slim Servers where access tokens can be requested for renewal by an application.

Access Token Table: This is a list of all access tokens granted by the SLIM server or installed with an application subscription.

When does the LSM operate?

An Application is Subscribed

When an application is subscribed, the Application Table, Application Server Table, and Slim table must be updated to reflect the new subscription. An expired token is added to the Access token database. The normal token renewal processes with update the token on the first installation, but the sleep loop for the token thread is interrupted so that the token renewal thread is woken up. The token renewal thread will see the expired token for the newly subscribed application and issue a renewal request to the SLIM Server.

Application subscription tasks:

1. Update the tables
2. Wake up the Access token Renewal thread.

A user Logs on

When a user logs on to the system the client will get a complete list of Application Identifiers numbers from the web server and compare them with the application table. If there are any applications that are no longer subscribed or new application the system does not know about a Message box will appear asking the user if they would like to update their subscription information. The synchronization of application sets from client to client is only handled by the client user interface. Another task started at logon time is the Token renewal thread. The token renewal thread scans the token list to see if any token is ready to be renewed. The token renewal thread is sleeps on a long timer and is only woken up if a new access token is added to the token database. When the sleep timer on the thread

expires, the thread will scan the list of access tokens to see if any of them have expired or are near expiration.

Logon Tasks:

1. Establish a logon with the ASP Web server. Get a list of all applications ID numbers and check if any have been added or deleted. If there are any adds or deletes force the user to go to the Client administration UI.
2. Update the Application Server Table
3. Update the Slim Server Table
4. Start the token renewal thread.

An application is started

When an application is started, the ECM will request an access token from the LSM. If the token is expired then a Message box will be presented to the user with information about the expiration of the application. The user can go from that User Interface to the client UI that administrates application subscriptions.

An Application is un-subscribed

When an application is un-subscribed, the corresponding records in the Application Table, Application Server Table, Slim table, and Access token table must be deleted.

Data type definitions

These tables are stored in an mdb database file. This database file can be accessed using ODBC or ADO database drivers.

Application Table

A master table of all application that are subscribed.

Name	Application ID	Application name	RootFileNumber	ForcedUpgrade	Message
Type	GUID 128 bit	String	128 bits	8 bits	String

This table describes the data in the eStreamAppUpgradeInfo Structure[1]. This table will link application names with Application ID numbers.

Application Server Table

This table will link Application Servers with subscribed Applications. This data is used by the ECM to get file information.

Name	Application ID	Application Server IP
Type	GUID 128 bit	IP Address 32 bit

A particular client application can have more that one server.

Slim Server Table

This table is a list of all servers where access tokens can be requested. An application can have more than one Slim Server.

Name	Application ID	Slim Server IP
Type	GUID 128 bit	IP Address 32 bit

[1] Software License and Management Server Amil Patel, Bhaven Avalandi, Michael Bechman, Sameer Panwar

Access Token Table

Name	ATID	UserID	AppID	Expiration Date
Type	GUID	GUID	GUID	Date

Access Tokens are requested when an application is first subscribed. They are renewed before they expire.

User Table

Name	User ID	User Name	Password Hash
Type	GUID	String	String

A users password is never saved. The user password is fed to a one way hash function and the hash output is what is saved to the database.

Interface definitions

Function 1

```
BOOL GetToken{
   GUID AppID,
   int size,
   eStreamAccessToken * TokenData);
```

Input:

GUID AppID: the Application ID for which a token is being requested.

Int size: the size of the token data structure

eStreamAccessToken * TokenData: the token structure.

Output:

The function will return True if a valid access token has been granted, false otherwise. The function will handle client UI for expired tokens.

Comments:

The eStreamAccess Token is defined in the slim server documentation.

Errors:

Application ID not found.

```
DWORD GetAppServer(GUID appID, int servernum)

Input:
   GUID AppID, this is the application ID for
   which a server is being requested.

   Int servernum, this number is usually 0, if
   a particular server is not available then
   the ECM will ask for the next server.  If
   the Server table does not have a server then
   this function will return -1

Output:

32 bit IP address

Comments:
A return code of -1 indicates that we have run
out of servers

Errors:

Application ID not found.
```

Component design

Database access

Access to the database tables is through either an ODBC or ADO interface. The AIM can access the databases directly though these interfaces. The program code that updates these tables is considered part of the AIM, but the databases themselves are considered part of the LSM.

Token Renewal Thread

The token renewal thread is a worker thread that is part of the Client executable. This thread is started with AfxBeginThread.

```
UINT MyThreadProc( LPVOID pParam )
{
  CTokenRenewalObject* pObject = (CTokenRenewalObject *)pParam;

  if (pObject == NULL ||
    !pObject->IsKindOf(RUNTIME_CLASS(CTokenRenewalObject)))
  return 1;  // if pObject is not valid

  // do something with 'pObject'

While(1)
{
        if(CheckForTermination())
                return 0;
        pObject->AccessTokenTable->MoveFirst()
        while(!pObject->AccessTokenTable->IsEOF())
        {
                if (NearOrAtExpiration(pObject->AccessTokenTable->Expiredate))
                {
                        pObject->RenewToken(pObject->AccessTokenTable->TokenID);
                }

                pObject->AccessTokenTable->MoveNext();
        }
```

```
        ComputeSleepTime();
        Sleep();
}


    return 0;  // thread completed successfully
}

// inside a different function in the program
.
.
.
pNewObject = new CMyObject;
AfxBeginThread(MyThreadProc, pNewObject);
.
.
```

Logon Initialization

This block of code is part of the Client executable when the client first starts up.

```
void ClientStartup(void)
{
   LogontoASPWeb();
   GetSlimServerTable();
   GetAppServerTable();

   StartTokenRenewalThread();

}
```

Testing design

The LSM is part of the Client Executable. This test plan will become part of the client executable test plan.

Unit testing plans

1. Initial Application Subscription Install. Does the AIM update the LSM tables correctly? This can be checked using Access.
2. Contact the ASP web server and Update the Application Server Table.
3. Contact the ASP web server and get a list of applications.
4. Does the Client UI respond when the list of applications is changed?
5. Force tokens to expire using Access and see if they are renewed by the Slim Server
6. Force access tokens to expire and prevent the Slim server from granting them.

Stress testing plans

Stress happens when Application Server and Slim Servers go down. Stress testing can be accomplished by running subscribed applications and then shutting down servers.

Coverage testing plans

Every application that eStream is capable of serving will need to be tested.

Cross-component testing plans

The LSM communicates with

ECM

AIM

Web Server

Slim Server

Client UI Components.

Upgrading/Supportability/Deployment design

This component is part of the client executable program. When the client executable program is upgraded then this component will be upgraded.

Open Issues

Every time the user logs on to the client a list of currently subscribed, applications must be downloaded from the server to insure client synchronization. Right now we don't have this function defined on the server side.

Omnishift C/C++ Coding Standard

[redacted]

Omnishift Confidential

The following proposal is based on the C++ coding standards document available at `http://www.possibility.com/Cpp/CppCodingStandard.html`. This document will concisely present the coding standards from the coding standard document. The reader should refer to the original document (linked above) for a detailed explanation of the standards. This document has the following sections:

NAMES: This section contains the naming schemes.
ERRORS AND ERROR CODES: This section contains the error formats and the error codes to be used by eStream 1.0 client and server components.
FORMATTING: Code layout and formatting guidelines.
COMMENTS: Guidelines for applying comments to the code.
LOOSE END: Loose ends.

Table of Contents:

NAMES

ID	RULE
Variables	• Use upper case as word separators, lowercase for the rest of the word. • No underbars ('_') • First letter of the variable is uppercase unless it is prepended with some other letters(see below). **Examples:** short Status; unsigned long TimeOfDay;
Pointers	• Prepend the variable with p. **Examples:** string* pName; char** ppValue;
Class Names	• Use upper case letters as word separators, lower case for the rest of a word • First character in a name is upper case • For externally exposed components, use the first 3 words to denote the component. • No underbars ('_') **Examples:** • class ConfigurationManager • class Config
Library Class Names	• Prefix the classname with OT **Examples:** • class OTHttpListener • class OTServer
Class Method Names	Same rule as for class names except for interfaces where the rule is: • Prefix the interface with the component's name. **Examples:** • ECMGetFileId • MonitorGetServerSet
Class Attribute Names	• Attribute names should be prepended with the character 'm'. • After the 'm' use the same rules as for class names. • 'm' always precedes other name modifiers like 'p' for pointer. **Examples:** int miLen; char* mpName; string* mpValue;
Global Variables	• Global variables should be prepended with "g". Examples: int gFlag; Logger gLog; Logger* gpLog;

Global Constants	• Global constants should be all caps with '_' separators. Examples: const int A_GLOBAL_CONSTANT= 5;
Type Names	• When possible for types based on native types make a typedef. • Typedef names should use the same naming policy as for a class with the word *Type* appended. Examples: typedef uint16 ModuleType; typedef uint32 SystemType;
#defines and Macros	• Put #defines and macros in all upper using '_' separators Examples: #define MAX(a,b) blah #define IS_ERR(err) blah
Function Names	• Use upper case letters as word separators, lower case for the rest of a word • First character in a name is upper case • No underbars ('_') **Examples:** • int SomeBloodyFunction()

Enum Names	• all upper using '_' separators
	Examples:
	enum PinStateType
	{
	PIN_OFF,
	PIN_ON
	}

File Names	• File should be all lower case • File name format should be <component>_<sub-component>.* **Examples:** monitor_heartbeat.cpp core_configmanager.c

ERRORS AND ERROR CODES

The following pound defines should be used for returning all successes and failures.

#define SUCCESSWITHINFO -1
#define SUCCESS 0
#define FAILURE >0 (The number representing an Error ID).

All error messages will be prepended with an error code. The format for error code will be as follows:

[ERRORID] [Severity] [Error Message]

where,

1. ERRORID are unique across the system.
2. Severity can be one of the following:
    a. 1-Low : A warning which can be ignored.
    b. 2-Medium: A warning which needs to be looked into.
    c. 3-High: Recoverable error in the component.
    d. 4-Critical: Irrecoverable error. Needs admin assistance.
3. The error message in itself should have the following format:
   [COMPONENT]:[ERROR MESSAGE]:[WORK AROUND]

Error Ids distribution for client and server are as follows:

0-1000 Server Internal Error Codes.
1001 – 8000 Server Error Codes.
8001 – 9000 Client Internal Error Codes.
9001 –16000 Client Error Codes.

FORMATTING

The following formatting policies should be followed by all code.

Braces Policy.

Place the braces in line with the keywords. An example of this is:

```
if ( 0 == a)
{
...
}
else
{
```

...
}

Indentation/Tabs/Space Policy

Use the standard Visual C++ settings which are(using Tools->Options->Tabs menu):

Indent Size: 4
Auto Indent: Smart
100 previous lines used for context.

White spaces should be spaces and NOT tabs.
VC++: Select "Insert Spaces" option. (This is NOT the default).
Emacs: Refer http://www.delorie.com/gnu/docs/emacs/emacs_205.html

Line Size etc.

1. Line size should not exceed 78 characters.
2. There should be one statement per line. The following piece of code violates this principle.

```
if (a>b) a++;
```

Method/Functions Formats.

1. Methods should preferably be less 50 lines of code.
2. Methods should not have more than 4 levels of nesting.
3. Methods should preferably be re-entrant. Non-reentrant methods should be clearly marked as such.
4. Each method/function should be preceded with a comment describing the method:

```
/*****************************************************************
FUNCTION:
INPUTS:
OUTPUTS:
DESCRIPTION:
ERRORS:
*****************************************************************/
```

COMMENTS

1. Every decision should have comments. The following keyword are associated with decisions:
    a. if, else
    b. while, continue
    c. switch, case, default, break
    d. goto
    e. return
2. Every class should have comment header with the following format:

```
/******************************************************************
CLASS NAME:
DESCRIPTION:
FRIEND CLASSES:
INCLUDES:
LIBRARIES:
******************************************************************/
```

3. Every function/method should have a header. (described above).
4. Every file should have a header describing the contents of the file.
5. Every directory should have a README describing the contents of the directory.
6. Make GOTCHAS explicit. Use the following format for gotchas.

- **:TODO: topic**
  Means there's more to do here, don't forget.
- **:BUG: [bugid] topic**
  means there's a Known bug here, explain it and optionally give a bug ID.
- **:KLUDGE:**
  When you've done something ugly say so and explain how you would do it differently next time if you had more time.
- **:TRICKY:**
  Tells somebody that the following code is very tricky so don't go changing it without thinking.
- **:WARNING:**
  Beware of something.
- **:COMPILER:**
  Sometimes you need to work around a compiler problem. Document it. The problem may go away eventually.
- **:ATTRIBUTE: value**
  The general form of an attribute embedded in a comment. You can make up your own attributes and they'll be extracted.

LOOSE ENDS

The following section notes some loose ends which do not fall in any of the categories above:

1. Always **initialize all variables** every time.
2. Use **header file guards** against multiple inclusions of the header file. The guards would look like:

   ```
   #ifndef ClassName_h
   #define ClassName_h
   ....
   #endif // ClassName_h
   ```
3. Object constructors should just initialize data. (They cannot return errors). Explicit Initialize() calls should be made to do any involved work.
4. Use **continue** and **goto** sparingly.
5. Be **"const" correct**. Use "const" wherever and whenever applicable.
6. All classes must have a **Default Constructor** and a **Copy Constructor**

QUESTIONS TO ASK AN ASP

ASP Overall

1. What is the current solution deployed by the ASP? Why?
2. What applications are most interesting to the ASP? What platforms do these applications run on?
3. What applications do they currently support? What are the licensing models for these applications? *
4. How often are the applications upgraded? *
5. What are the bottlenecks faced with the current solution?
6. How is user data handled? Is there an issue with the handling of user data?
7. What billing systems does the ASP use? *
8. What pricing models does the ASP employ? What pricing models are important to the ASP for the future? *
9. What is the growth estimate in terms of the number of users?
10. Who are the ASP's key partners?
11. Who are the ASP's key suppliers (or dependencies)?
12. Who are the ASP's key competitors?
13. How does the ASP differentiate its products?
14. Does the ASP have a QOS solution?
15. What are the ASP's major security concerns?
16. What is limiting the ASP's growth?
17. What are the ASP's current bandwidth requirements?
18. How significant an operating cost is this expense?
19. How does the ASP expect this bandwidth requirement to change over time?

ASP Infrastructure

1. What are the platforms on which the ASP applications run?
2. What are the platforms on which the ASP servers run? *
3. Does the ASP maintain the infrastructure? If not, where is it outsourced and why?
4. What are the current data center space needs?
5. How significant an operating cost is this expense?
6. How does the ASP expect this space need to change over time?
7. What kinds of load balancing systems does the ASP deploy? *
8. What are the app-servers (middleware) used by the ASP?
9. What ports do the servers use in the current solution? *
10. How many users does each server adequately support?
11. What are the most important features (wish list) for the app-server infrastructure?
12. Is the ASP willing to add h/w accelerators for encryption?
13. How much unplanned down time has the ASP been experiencing?
14. What have been the main causes of these down times?

ASP end user

1. Is it ok to ask the user to do reboots? *
2. How big of a cache can eStream assume? *
3. Does eStream need to support local installations of the same application? *
4. What is the minimum performance tolerated by the end user?
5. What is the typical bandwidth experienced by the end user?
6. Is the end user always connected?
7. Is there a need to allow for offline access of applications?
8. Can eStream assume that the end user has the ability to install drivers?
9. Is the end user always behind firewalls?
10. What is the accepted level of security for the client-server communication? *
11. What is the accepted level of piracy protection needed on the client side? Does the ASP need its own Intertrust kind of solution? *
12. What is the acceptable delay for installing an application? (Basically, what is the maximum size of the AppInstallBlock acceptable to the ASP?) *
13. Does the ASP need an ability to terminate end users in the middle of a billing cycle? *
14. Can the end user run the same application on multiple clients at the same time? *
15. If the license expires, can the client continue running if the application is in the cache? *
16. Is collecting profile data from the end user acceptable? Is the profile data of use to the ASP?

Deployment

1. What makes for a "good" deployment?
2. Describe a "good" deployment?
3. How would the ASP like to work with a solution provider to deploy their solution in the ASP environment? (i.e. solution provider installs for ASP; help the ASP install; ASP installs alone, other)
4. How would the ASP look to deploy a solution like eStream? (i.e. phased implementation; full-blown roll-out, other)
5. How would the ASP want to receive the eStream solution software? (i.e. physically, electronically)
6. What things would be of concern to the ASP in deploying eStream in the ASP environment?
7. What sort of technical questions would the ASP want answered before considering deploying a solution like eStream?
8. How would the ASP want to receive training on the solution and it's installation and administration?
9. Would there be a need for a Certification Program for eStream Administrators?
10. What sort of tools/abilities would the ASP require to manage and monitor a solution like eStream?
11. What kind of monitoring solutions does the ASP IT team use?

Support

1. What would the ASP like to see/require in an SLA?
2. What makes for a "good" support experience?
3. Who, that the ASP is currently working with, provides "good" support and what happens?
4. What is missing in the current SLA's?
5. What would the ASP like to see/require in support provided by a solution provider?
6. What is missing from current solution provider support offerings?
7. To what extent would the ASP work with a solution provider in remotely troubleshooting problems?
8. Would allowing us remote access into a "Monitor" tool/DB or Log running on the ASP system to effectively troubleshoot a problem be acceptable? How would the ASP see this working?
9. Would the ASP provide first line support to the end users?
10. What does the end user support offering consist of?
11. What would the ASP require from us in order to be able to offer end-user support?
12. Does the ASP actually provide that support or does it have a 3rd party do that on its behalf?
13. How would the ASP like to be notified of upgrades and patches?
14. How would the ASP like to receive upgrades and patches?

Citrix

1. What does the ASP like about MS Terminal Services and Citrix offerings?
2. What does the ASP NOT like about MS Terminal Services and Citrix offerings?
3. What 3rd party tools does the ASP use in addition to or in place of Citrix's offerings and why?
4. How well does the Citrix solution scale? Perform?
5. How would the ASP rate Citrix's support offerings?
6. If the ASP were redesigning Citrix Support what would be done differently?
7. How did the ASP deploy the Citrix solution?
8. What sort of issues did the ASP run into in deploying the Citrix solution?

* - Affecting current design

Software Development Process

Version 0.2

Ricky Benitez

[redacted]

We are what we repeatedly do,
Excellence, then, is not an act, but a habit.

ARISTOTLE

This document describes the software development process followed at OTI for all software products. The purpose of this process is to ensure that software products are developed in an effective, predictable, repeatable and continuously improvable manner. The process has inputs, methods, outputs and metrics to determine its effectiveness.

Ownership

The overall responsibility for the development, refinement, effectiveness and adherence to the software development process belongs to the VP of Engineering. Responsibility for the various steps in the process will be assigned to appropriate individuals depending on the scope of the product and the makeup of the development group.

Inputs

The inputs to the software development process are:

1. Marketing Requirements Document (MRD)
2. Technology White Papers and Prototypes

Marketing Requirements Document

The VP of Marketing is responsible for the development of this document. The VP of Engineering is responsible for extracting the portions of the MRD that will be implemented in software and capturing these as the High Level Requirements document.

Technology White Papers

The Chief Technology Officer is responsible for providing any technology white papers and prototypes that provide proof of concept and suitable technological direction needed to convert the marketing requirements into a software product.

Methods

The primary methods used during the development process are described in detail later in the document, but primarily consist of:

1. Transforming the MRD to the High Level Requirements document (HLR)
2. Transforming the HLR to the Product Datasheet (PDS)

3. Transforming the HLR to the High Level Design document (HLD)
4. Transforming the HLR to the Product Test Plan (PTP)
5. Transforming the HLD to the Low Level Design documents (LLDs)
6. Transforming the HLD to an End-to-end Test Infrastructure (ETI)
7. Transforming the LLDs to an Integration Test Infrastructure (ITI)
8. Transforming the LLDs to Product Source Code (PSC)
9. Integration
10. Final Validation
11. Performing a Post Mortem

Outputs

The outputs of the software development process are:

1. Updated MRD
2. Product Datasheet (PDS)
3. Digital components that meet the requirements listed in the HLR
4. A set of mutually consistent and appropriately labeled design documents, source code, build environment, build infrastructure, test plans and test harnesses in a revision control system
5. An issue database which contains a description of all past and outstanding issues relating to the requirements, design and implementation of the software components of the product
6. A post mortem report indicating what was learned during the development process that can be used to improve the process in the future.

Metrics

Metrics are needed to track the progress of the development process and to constantly improve and fine-tune it. The primary mechanisms employed to measure are:

1. Completion of the methods
2. The product issue database
3. The process issue database

These will be discussed in detail later in this document.

Overview of the Process

The software development process is iterative. While it may be described in a linear fashion, it must be understood that events will often dictate that previously completed portions of the process need alterations and that these alterations may propagate forward or backward along the various stages of the process. The ultimate objective is to end up with a set of outputs that are consistent relative to each other and relative to the inputs.

Supporting Documents

The following documents exist in conjunction with this one to support and fully define the overall software development process:

1. Coding Guidelines
2. Configuration and Release Management Guidelines

Description of the Methods

MRD to HLR

The input of this method is a marketing requirements document and its output is a high-level requirements document. This method is performed once on a static MRD and then iterates incrementally as events warrant. Every change to the MRD after the initial HLR is produced generates an issue. The HLR must be kept in synch with the MRD and vice-versa. Once the initial HLR is produced changes are made only as a step towards resolving an issue.

Description

The HLR is a set of precise imperatives that collectively define the scope and behavior of the software product. Each precise imperative is known as a requirement. Requirements are grouped into the following categories:

1. Functionality
2. Localization
3. Usability
4. Reliability
5. Performance
6. Scalability
7. Security
8. Portability
9. Maintainability

Every product will have at least one requirement within each of these categories. Each requirement consists of the following:

1. Unique number – used to identify the requirement in other documents
2. Description – a concise description of the requirement
3. Importance – the level of importance to the final product between 1 and10. An importance of 1 indicates that the failure to fully or partially meet that requirement will have very minimal impact on the success of the product. An importance of 10 indicates that the product must either be able to meet the specified requirement or should not be produced at all

Use cases are also included in an HLR to describe the various scenarios that the product is expected to handle from the perspective of its various users. Use cases are presented with the following information:

1. Summary – a description of the use or activity
2. Actors – a listing of those who have a role in the activity
3. Inputs – the initial state of the product and additional data that the actors need to [illegible]
4. Processing – the sequence of steps taken by the actors
5. Output – the final state of the system and any data that the system produces

Template

An HLR template is located in the OTI share under \\fserv1\oti\general\templates.

Review

Upon completion of the HLR, a requirements review will be held. The requirements review process is as follows:

1. No less than four primary reviewers are selected, one should be from the engineering team (preferably the architect of the system), one should be from product marketing, one should be from the release group and one should be from the deployment or PSO organization.
2. The primary reviewers are given a copy of the MDR and the HLR document sufficiently prior to the review meeting to review the HLR.
3. The primary reviewers ensure to the best of their ability that:
    - The requirements are precise
    - There are appropriate requirements for every category (or a good understanding as to why no requirement is needed for a particular category)
    - The requirements and their importance level will, if faithfully transformed into a software product and incorporated with appropriate marketing, deployment and support, result in a successful customer solution
4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the requirements.
5. If the requirements have not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.
8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
    - HLR is completed after all issues are satisfactorily resolved
    - HLR must be reviewed again after all issues are satisfactorily resolved
    - HLR must be abandoned and a new requirements effort begun
9. After the meeting, the scribe is responsible for filing and submitting the review report. If the HLR is not to be abandoned, the scribe is also responsible for entering all issues into the issue tracking system.

Completion

The MDR to HLR method is considered complete only after:

- a review of the HLR whose primary reviewers decide should be considered completed has taken place
- a review report has been submitted for every review of the HLR
- all HLR issues have been submitted and satisfactorily resolved
- the HLR has been appropriately added to the revision control system

Issue Resolution

HLR issues that result in the removal, change or inclusion of one or more requirements can be considered resolved after the HLR and any other document that are affected and must be kept in synch with the HLR are appropriately updated and each change has been reviewed and approved by one reviewer from the engineering team, one from the marketing team, one from the product marketing group, one from the release group and one from the deployment or PSO group. If the HLR is significantly changed as a result of addressing an issue, a full review should be considered.

HLR to PDS

The input to this method is a high-level requirements document and the output is a product datasheet. This method is performed once on the HLR and then iterates incrementally as events warrant. The PDS must be kept in synch with the HLR and vice-versa. Once the initial PDS is produced changes are primarily made only as a step towards resolving an issue. The speculative nature of the PDS gives its owner more leeway during review and editing than is accorded to the owners of most other product documents and deliverables.

Description

The PDS is a document consisting of various sections that describe the overall software product to potential stakeholders. The PDS contains a section for each of the following:

1. Is/Is Not – this section describes the product in terms of what it is and, to dispel potential misconceptions about what it might be, what it is not
2. Flexibility Matrix – this section indicates the relative importance of time, features and cost
3. Requirements – this section is a copy of the HLR, potentially tailored to a less technical audience
4. Schedule – this section lists the major milestones of the product
5. Dependencies – this section lists the major external dependencies of the product
6. Resources – this section lists the resources needed by engineering to complete the product as follows:
    a. Engineers and managers needed per month
    b. Machine resources needed
    c. Software resources (licenses, etc.) needed
7. Quality Plan – this section lists the steps that will be taken towards reaching the quality goals of the product:
    a. What will be minimally prototyped prior to high-level design
    b. Any exceptions or additions to the software development process defined in this document
    c. Coverage, number of white-box tests and other criteria required to meet completion on PSC and final validation
    d. Alpha, beta and other pre-production releases
8. Risks – this section lists the anticipated risks and appropriate contingency plans to address those risks.

Template

A PDS template is located in the OTI share under \\fserv1\oti\general\templates.

Review

Upon completion of the PDS, it is posted for the managers, engineers and stakeholders of the product to review.

Completion

The HLR to PDS method is considered complete only after:

- The PDS has been completely filled to the best understanding of its owner
- The PDS has been posted for review

Issue Resolution

PDS issues that result in changes to the PDS are considered resolved after the PDS and any other document affected that must be kept in synch with the PDS are appropriately updated and reviewed.

HLR to HLD

The input of this method is a high-level requirements document and the output is a high-level design document. This method is performed once on the HLR and then iterates incrementally as events warrant. The HLD must be kept in synch with the HLR and vice-versa. Once the initial HLD is produced changes are made only as a step towards resolving an issue.

Description

The HLD is a document consisting of various sections that describe the overall software product in high-level form. The HLD contains a section for each of the following:

1. Definition of terms – this section defines each technical term that may lead to confusion in the understanding of the design if left undefined
2. Block diagram – a pictorial description of the system, to aid in the identification and understanding of the components and APIs described in the design
3. High-level description of each component – a description of each major component of the system which clearly describes the role that each component plays in the overall system and serves as the launching point of the component's low level design
4. High-level description of each API – a catalog and simple description of each major API in the system, which a particular emphasis on the system's external APIs
5. High-level test strategy – a description of the approach to be taken to validating the correctness of the system end-to-end

Template

An HLD template is located in the OTI share under \\fserv1\oti\general\templates.

Review

Upon completion of the HLD, a low-level design review will be held. The HLD review process is as follows:

1. No less than two primary reviewers are selected.
2. The primary reviewers are given a copy of the HLR and the final HLD document sufficiently prior to the review meeting to review the HLD.
3. The primary reviewers ensure to the best of their ability that:
   - The design meets its requirements
   - All sections of the design are complete
   - The design is as simple, robust, testable, scalable and well thought through as time requirements permits
   - Competent designers could perform a low-level design of each component given the HLR and HLD without needing to consult the author of the HLD
4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the design.
5. If the design has not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.
8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
   - Design is completed after all issues are satisfactorily resolved
   - Design must be reviewed again after all issues are satisfactorily resolved
   - Design must be abandoned and a new design effort begun
9. After the meeting, the scribe is responsible for filing and submitting the review report. If the design is not to be abandoned, the scribe is also responsible for entering all issues into the issue tracking system.

Completion

The HLR to HLD method is complete only after:

- a review of the HLD whose primary reviewers decide should be considered completed after all issues are resolved has transpired
- a review report has been submitted for every review of the HLD
- all HLD issues have been submitted and satisfactorily resolved
- any changes required to the HLR to bring both HLR and HLD documents in synch have been submitted as issues
- the HLD has been appropriately added to the revision control system

Issue Resolution

HLD issues that require changes to the HLD can be considered resolved after the HLD document and any other documents that are affected and must be kept in synch with the HLD are appropriately updated and each change has been reviewed and approved by at

least one reviewer. If the HLD is significantly changed as a result of addressing an issue, a full review should be considered.

HLR to PTP

The input to this method is the high-level requirements document and the output is a product test plan. This method is performed once on a static HLR and iterates incrementally as events warrant. The PTP must be kept in synch with the HLR and vide-versa. Once an initial PTP is produced changes are made only as a step towards resolving an issue.

Description

The PTP is a document describing the various black-box and stress tests that will be applied to the product to ensure that it meets its stated requirements. The PTP should completely cover every requirement in the HLR to ensure that the delivered product meets its stated goals.

Template

TBD

Review

Upon completion of the PTP, a test plan review will be held. The PTP review process is as follows:

1. No less than two primary reviewers are selected.
2. The primary reviewers are given a copy of the final PTP document sufficiently prior to the review meeting to review the PTP.
3. The primary reviewers ensure to the best of their ability that:
    - There are appropriate tests defined to cover all the requirements
    - The test are as simple, robust, repeatable, scalable and well thought through as time requirements permits
    - A competent SQA engineer could conduct the test using the PTP without needing to consult its author
    - The PTP meets the quality criteria stated in the quality plan section of the PDS
4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the test plan.
5. If the test plan has not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.
8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
    - Plan is completed after all issues are satisfactorily resolved

    - Plan must be reviewed again after all issues are satisfactorily resolved
    - Plan must be abandoned and a new planning effort begun
9. After the meeting, the scribe is responsible for filing and submitting the review report. If the plan is not to be abandoned, the scribe is also responsible for entering all issues into the issue tracking system.

Completion

The HLR to PTP method is complete only after:

- a review of the PTP whose primary reviewers decide should be considered completed after all issues are resolved has transpired
- a review report has been submitted for every review of the PTP
- all PTP issues have been submitted and satisfactorily resolved
- any changes required to the HLR to bring this documents in synch with the PTP have been submitted as issues
- the PTP has been appropriately added to the revision control system

Issue Resolution

PTP issues that require changes to the PTP can be considered resolved after the PTP document and any other documents that are affected and must be kept in synch with the PTP are appropriately updated and each change has been reviewed and approved by at least one reviewer. If the PTP is significantly changed as a result of addressing an issue, a full review should be considered.

HLD to LLDs

The input to this method is a high-level design document and the output is one low-level design document for each component described in the HLD. This method is performed once on the HLD and then iterates incrementally as events warrant. The LLD must be kept in synch with the HLD and vice-versa. Once the initial LLD is produced changes are made only as a step towards resolving an issue.

Description

The LLD is a document consisting of various sections to describe the low-level design of a component. The LLD contains a section for each of the following:

1. Functionality – this section describes the functionality that this component provides
2. Data type definitions – a list of the data types defined or used by this component
3. Data structures – a description of the data structures defined and used by this component
4. Interface definitions – a description of the external and internal interfaces supported by this component — for user interfaces, a mock-up or prototype of the interface must be provided (separate from the LLD) and appropriate screen shots of this prototype should be included
5. Component design description – a description of the component sufficient to allow a competent programmer to implement the component without needing to consult the original author

6. Testing – a description of the test harnesses used to unit test, stress test and/or coverage test this component sufficient to allow a competent programmer to implement these harnesses without consulting the author
7. Supportability – a description of the supportability features of the component such that an integration engineer can make full use of the supportability features built into the component without needing to consult the author or developer

Template

An LLD template is located in the OTI share under \\fserv1\oti\general\templates.

Review

Upon completion of the LLD, a low-level design review will be held. The LLD review process is as follows:

1. No less than two primary reviewers are selected.
2. The primary reviewers are given a copy of the final LLD document sufficiently prior to the review meeting to review the LLD.
3. The primary reviewers ensure to the best of their ability that:
    - The design meets its requirements
    - All sections of the design are complete
    - The design is as simple, robust, testable, scalable and well thought through as time requirements permits
    - A competent software engineer could implement the component using the LLD without needing to consult its author
4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the design.
5. If the design has not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.
8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
    - Design is completed after all issues are satisfactorily resolved
    - Design must be reviewed again after all issues are satisfactorily resolved
    - Design must be abandoned and a new design effort begun
9. After the meeting, the scribe is responsible for filing and submitting the review report. If the design is not to be abandoned, the [illegible] is also responsible for entering all issues into the issue tracking system.

Completion

The HLD to LLD method is complete only after:

- a review of the LLD whose primary reviewers decide should be considered completed after all issues are resolved has transpired

- a review report has been submitted for every review of the LLD
- all LLD issues have been submitted and satisfactorily resolved
- any changes required to the HLD to bring this document in synch with the LLD have been submitted as issues
- the LLD has been appropriately added to the revision control system

Issue Resolution

LLD issues that require changes to the LLD can be considered resolved after the LLD document and any other documents that are affected and must be kept in synch with the LLD are appropriately updated and each change has been reviewed and approved by at least one reviewer. If the LLD is significantly changed as a result of addressing an issue, a full review should be considered.

HLD to ETI

The input to this method is a high-level design document with a high-level test strategy description and the output is an end-to-end test infrastructure. This method is performed once on the HLD and then iterates incrementally as events warrant. The ETI infrastructure must be kept in synch with the high-level test strategy in the HLD and vice-versa. Once an initial ETI is produced changes are made only as a step towards resolving an issue.

Description

The ETI is a set of scripts, source files, include files, make files and documents that collectively are used to automate the process of performing a rigorous end-to-end test of the integrated PSC. The purpose of the ETI is to provide an automated method for ensuring that the overall integrated PSC operates on a regular (nightly) basis.

Template

TBD

Review

Upon completion of the ETI, a code review will be held. The code review process is as follows:

1. No less than two primary reviewers are selected.
2. The primary reviewers are given a copy of the ETI and related code (or pointed to their location in the configuration management system) sufficiently prior to the review meeting to review the code.
3. The primary reviewers ensure to the best of their ability that:
   - The code is an efficient, reasonable and complete implementation of the high-level test strategy. Efficiency means that the end-to-end test does not require manual steps and can complete in no more than a couple of hours.
   - A competent SQA engineer could maintain the ETI without needing to consult its author (assuming that the HLD is also available)

4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the implementation.
5. If the implementation has not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.
8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
   - Implementation is completed after all issues are satisfactorily resolved
   - Implementation must be reviewed again after all issues are satisfactorily resolved
   - Implementation must be abandoned and a new implementation effort begun

After the meeting, the scribe is responsible for filing and submitting the review report. If the implementation is not to be abandoned, the scribe is also responsible for entering all issues into the issue tracking system.

Completion

The HLD to ETI method is complete only after:

- a review of the ETI whose primary reviewers decide should be considered completed after all issues are resolved has transpired
- a review report has been submitted for every review of the ETI
- all ETI issues have been submitted and satisfactorily resolved
- any changes required to the HLD test strategy to bring this documents in synch with the ETI have been submitted as issues
- the ETI has been appropriately added to the revision control system as specified in the configuration and release management guidelines
- the ETI has been set up to run automatically on each successful and sanity-tested build of the integrated PSC and the results of the test are being mailed to all interested parties

Issue Resolution

ETI issues that require changes to the test infrastructure must be appropriately reviewed before they are closed. The review process depends on the nature of the changes made to the ETI. If only one to twenty-five source statements are affected, then a desk check review performed by another member of the technical staff is all that is required. That member needs to add their review comments to the issue prior to closing it. If more than twenty-five statements are added or modified, then the standard review process for ETI must be followed.

LLD to ITI

The input of this method is the set of low-level design documents with component test strategies and the output is an integration test suite definition and infrastructure. This method is performed once after all or after a substantial portion of the LLDs have been produced and then iterates incrementally as events warrant. The ITI must be kept in synch with the LLDs and vice-versa. Once the initial ITI is produced changes are made only as a step towards resolving an issue.

Description

The ITI is a set of scripts, source files, include files, make files, test definition files and documents that collectively are used to automate the process of performing a rigorous set of regression, black-box, stress, coverage, performance and sanity tests on the PSC. The purpose of the ITI is to provide an automated method for ensuring that the individual product components and the overall integrated PSC is robust on a regular (nightly) basis.

Template

TBD

Review

Upon completion of the ITI, a test suite and code review will be held. The code review process is as follows:

1. No less than two primary reviewers are selected.
2. The primary reviewers are given a copy of the test suite and related code (or pointed to their location in the configuration management system) sufficiently prior to the review meeting to review the suite and the code.
3. The primary reviewers ensure to the best of their ability that:
    - The test suite will efficiently provide a high-level of confidence that every component of the PSC is of high quality. Efficiency means that the entire ITI runs in no more than a few hours. This entails examining each test suite and ensuring that tests are not overly redundant or spend inordinate amounts of time running while providing only marginal improvements in the confidence level of the product's quality.
    - The code is reasonable and complete automation of the integration test suite
    - A competent SQA engineer could maintain the ITI without needing to consult its author (assuming that the LLDs are also available)
4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the implementation.
5. If the implementation has not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.

8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
    - Test suite and implementation is completed after all issues are satisfactorily resolved
    - Test suite and implementation must be reviewed again after all issues are satisfactorily resolved
    - Test suite and implementation must be abandoned and a new implementation effort begun

After the meeting, the scribe is responsible for filing and submitting the review report. If the test suite and implementation is not to be abandoned, the scribe is also responsible for entering all issues into the issue tracking system.

Completion

The HLD to ITI method is complete only after:

- a review of the ITI whose primary reviewers decide should be considered completed after all issues are resolved has transpired
- a review report has been submitted for every review of the ITI
- all ITI issues have been submitted and satisfactorily resolved
- any changes required to the LLD test strategies to bring this documents in synch with the ITI have been submitted as issues
- the ITI has been appropriately added to the revision control system as specified in the configuration and release management guidelines
- the ITI has been set up to run automatically after each successful build of the integrated PSC and the results of the test are being mailed to all interested parties

Issue Resolution

ITI issues that require changes to the test suite or the source code must be appropriately reviewed before they are closed. The review process depends on the nature of the changes made to the ITI. If changes are one to several additional tests added to an existing test suite, then a desk check of each added test by another member of the technical staff is all that is required. If a new test suite or more than several new test cases are being added, then the standard review process for the ITI must be followed. If changes are to source code and only one to twenty-five source statements are affected, then a desk check review performed by another member of the technical staff is all that is required. That member needs to add their review comments to the issue prior to closing it. If more than twenty-five statements are added or modified, then the standard review process for ITI must be followed.

LLD to PSC

The input to this method is a low-level design document and the output is product source code. This method is performed once for each completed LLD and then iterates incrementally as events warrant. The LLD must be kept in synch with the PSC and vice-versa. Once the initial PSC is produced changes are made only as a step towards resolving an issue.

Description

The PSC is a collection of the source files, include files, make files and a collection of development tools and an environment that will be used to produce the binary bits that will be ultimately delivered to a customer. Additional supporting code including, but not limited to code generators, component tests, white box tests, regression tests and test drivers are produced while applying this method. PSC must adhere to the coding guidelines and the completion criteria for this method. PSC and all other supporting code must adhere to the configuration and release management guidelines.

Template

See the coding guidelines and configuration and release management guidelines for template information.

Review

Upon completion of the PSC, a code review will be held. The code review process is as follows:

1. No less than two primary reviewers are selected.
2. The primary reviewers are given a copy of the PSC and related code (or pointed to their location in the configuration management system) sufficiently prior to the review meeting to review the code.
3. The primary reviewers ensure to the best of their ability that:
    - The code is a reasonable and complete implementation of the LLD
    - The associated white-box and component tests specified in the LLD have been implemented and have passed successfully
    - The PSC adheres to the coding guidelines
    - A competent software engineer could maintain the PSC without needing to consult its author (assuming that the LLD and HLD are also available)
4. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the implementation.
5. If the implementation has not been fully reviewed by the primary reviewers, the review must be rescheduled.
6. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
7. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.
8. The scribe then polls the primary reviewers to determine which of the following courses of action will be taken:
    - Implementation is completed after all issues are satisfactorily resolved
    - Implementation must be reviewed again after all issues are satisfactorily resolved
    - Implementation must be abandoned and a new implementation effort begun

After the meeting, the scribe is responsible for filing and submitting the review report. If the implementation is not to be abandoned, the scribe is also responsible for entering all issues into the issue tracking system.

Completion

The LLD to PSC method is complete only after:

- a review of the PSC and associated code whose primary reviewers decide should be considered completed after all issues are resolved has transpired
- a review report has been submitted for every review of the PSC
- all PSC and associated code issues have been submitted and satisfactorily resolved
- any changes required to the LLD to bring this documents in synch with the PSC have been submitted as issues
- the PSC and associated code has been appropriately added to the revision control system as specified in the configuration and release management guidelines
- the PSC meets the coverage and component testing criteria specified in the quality plan section of the PDS

Issue Resolution

PSC issues that require changes to the source code must be appropriately reviewed before they are closed. The review process depends on the nature of the changes made to the PSC. If only one to twenty-five source statements are affected, then a desk check review performed by another member of the technical staff is all that is required. That member needs to add their review comments to the issue prior to closing it. If more than twenty-five statements are added or modified, then the standard review process for PSC must be followed.

Integration

The input to this method is the separate components product source codes, the integration test infrastructure and the end-to-end test infrastructure and the output is the fully integrated product source code. This method is performed iteratively starting as soon as there is more than one interacting component PSC available and as long as there are changes made to any of the component PSC. The method must take place as frequently as is practically possible while the PSC is changing. The desired goal is once per day.

Description

Integration is the process of bringing all of the PSC components together to produce the entire product and validating them against a suite of regression, black-box, stress, performance, sanity tests and the ETI. A successful integration is a potential candidate for final validation. In order to provide developers with timely feedback concerning any integration and ETI failures that are introduced into the PSC, the integration process should be automated and performed as regularly as it is practical (every night, assuming that changes have been made to the PSC since the last successful integration).

Template
TBD

Review
Integration is by definition reviewed every time it is attempted (about once a day).

Completion
Integration is complete and a candidate for final validate is produced when:

- The entire system has been successfully built from the completed PSC of every component using only the documented build environment
- The built system successfully passes all regression, black-box, stress, performance, sanity and ETI test suites designated for integration
- All high and medium priority requirements, design and implementation issues have been resolved

Issue Resolution
There are rarely any issues against the integration phase. Most issues are ITI and ETI issues and are covered under those particular items. Issues might come up, however, in which integration is not taking place with the level of efficiency or producing the expected amount of confidence. In such cases, the issue may be categorized as an integration issue as a placeholder for determining whether it should be filed against ITI or ETI.

Final validation
The input to this method is a successful integration that produces a valid final validation candidate and the output is a fully validated release candidate. This method is applied as often as required, although it tends to be planned to coincide with code freeze and change control to ensure a quick and timely resolution of just those issues that are preventing the successful final validation of the product.

Description
Final validation is the process of taking a valid final validation candidate and applying the entire PTP against it. If every test defined in the PTP is successfully passed, then the candidate is declared to be a valid release candidate. Once this occurs, the build environment and all of the source files, include file, make files, test definition files and scripts that were used to produce the release candidate are appropriately labeled as stated in the configuration and release management guidelines.

Template
TBD

Review
Final validation is by definition reviewed every time it is attempted.

Completion

The integration stage is complete and a candidate for release is produced when:

- The entire PTP is successfully applied against a valid final validation candidate.
- The build environment and all of the source files, include file, make files, test definition files and scripts that were used to produce the release candidate are appropriately labeled as stated in the configuration and release management guidelines.
- No requirements, design, implementation or other types of issues logged against the current release with a priority greater than that stated as a requirement for release in the PDS remain unresolved.

Issue Resolution

There are rarely any issues against the final validation phase. Most issues are PTP issues and are covered under that category.

Post Mortem

The input to this method is a complete cycle through the software development process and the opinions of the participants. The post mortem method is applied once for each completed product release. The output is a post mortem report and a set of issues submitted against the software development process and any of the other supporting processes (such as the configuration and release management process).

Description

A post mortem captures the learning that took place over the software development lifecycle and should produce a number of issues against the software development process and other related processes that are to be resolved. If this is done soon after the completion of the development and release cycle and issues are addressed, then there is opportunity for continual refinement and improvement of the software development process and of the organization's development capability. The method requires that each participant fill out a feedback form and that these forms be reviewed by a set of individual who produce a post mortem report and file the issues that the reviewers determined were relevant against each appropriate process.

Template

TBD

Review

Upon completion of the post mortem feedback forms, a post mortem review will be held. The review process is as follows:

1. No less than two primary reviewers are selected.
2. All individuals who were asked to give feedback are invited to the review.
3. The primary reviewers are given a copy of the feedback forms sufficiently prior to the review meeting to review the code.
4. The primary reviewers ensure to the best of their ability that:

- All feedback comments are considered and appropriate root causes of issues are identified
- The issue tracking system is consulted to collaborate or disprove potential issues when appropriate

5. At the start of the review, a scribe (not a primary reviewer) is selected. The scribe notes down who is in attendance and ensures that all primary reviewers are present and have reviewed the implementation.
6. If the feedback has not been fully reviewed by the primary reviewers, the review must be rescheduled.
7. Primary reviewers bring up their issues and a decision is made as to whether the item is or is not a real issue. The scribe logs all real issues.
8. When the primary reviewers have presented all their issues, other attendees are invited to bring up issues. Real issues are logged.

After the meeting, the scribe is responsible for filing and submitting the post mortem report and entering all issues into the issue tracking system.

Completion

The post mortem is not complete until:

- Post mortem feedback has been received from all available participants
- The post mortem review is held
- All issues have been submitted against the appropriate process
- A post mortem report has been submitted

Issue Resolution

All issue filed against the software development process after the post mortem must be resolved in a timely manner. All changes must be reviewed and approved by the VP of Engineering.

Description of Metrics

TBD

//develop/eng/docs/readme.txt

This directory is for keeping all engineering related documents,
but not product specific documents. Product specific documents are
to be created with the product <docs> directory.

eStream 1.0 Requirements

Version 1.5

[redacted]

1.0 Introduction

This document describes the high level requirements for the eStream 1.0 product. These requirements are given first as lists for the client and server components and then as scenarios.

To facilitate the development of follow-on products, eStream 1.0 does not include attributes that explicitly preclude future support of thin clients or of data ubiquity.

2.0 Client Requirements

The following are performance and functional requirements for the client portion of eStream 1.0:

ID	Description	Priority
1.0	eStream client software operates on Windows 2000, Windows NT4, & Windows 98 for x86 clients.	10
1.1	eStream client software may collect profile information during application execution; this information is used to improve client-based prefetching. This data is not uploaded.	8
1.2	eStream client software supports eStream execution of top-selling (as represented by Ziff-Davis suites) desktop & laptop applications; these applications are listed in section 5.0 below. Other applications are supported (opportunistically) as well.	10
1.3	eStream client software is obtained via the web or via some distribution media & is installed via some industry-standard [e.g., installshield] mechanism; its installation requires administrative privileges. Install reboot should be avoided, unless needed to minimize potential stability/reliability problems. eStream client software can be upgraded w/o reinstallation and w/o breaking installed apps.	10
1.4	eStream client software operates across the different natural languages supported by Windows. The first release is an English-language release.	8
1.5	Applications running under eStream have an ave	8

	interactive response time within 10% of client native for connections at 256K bps or higher.	
1.6	eStream client software is able to operate with only 16M of available disk space; this is the minimum supported configuration. User is encouraged to allow cache to grow beyond an arbitrary limit for best performance.	8
1.7	eStream client software supports simultaneous execution of multiple eStreamed applications, including multiple instances of a single application (by a single user at a time; please see 1.21).	10
1.8	eStream client software is able to unambiguously reference a particular ASP license for an application.	10
1.9	Applications being eStreamed function in the same way that they would if they were installed locally.	10
1.10	eStream client software tolerates server failure [i.e., it continues running any active apps and allowing apps to be launched], though possibly with some delay, assuming that an alternative server of the needed type is accessible.	10
1.11	eStream client software detects and tolerates lost or garbled messages.	10
1.12	It is difficult to steal an eStream application's code or data from the client.	10
1.13	When an eStream application is being installed on a client, the process detects if the app is already installed & requests user confirmation to continue; a single version of an application is available to the user at a time. Install reboot should be avoided, unless needed to minimize potential stability/reliability problems.	10
1.14	Upon uninstalling an application, application-specific changes to the client system are removed or undone.	9
1.15	eStream client software makes minimal changes to the client system when running, avoiding/hiding any registry, DLL, & non-Z file system changes as needed.	10
1.16	eStream client software makes a run/no run license decision quickly enough when an eStreamed application is started not to cause customer satisfaction issues.	10
1.17	eStream client software is launched/terminated at boot/shutdown. It is normally activated/deactivated at logon/logoff of an eStream user; it may	10

	temporarily be deactivated/activated at other times to allow specific administrative activities to occur.	
1.18	User is able to set initial size of client cache. User is able to increase the size of the cache later without significant performance penalty.	9
1.19	eStream client software does not include explicit ASP logon/logoff to run installed apps; ASP identification data stored on client machine allows AccessToken to be obtained in seamless manner.	9
1.20	eStream client software facilitates roaming (i.e., moving one's client system to a different site); eStream server info is not invalid/inappropriate when the client is moved to a different venue.	10
1.21	Only one user at a time on a particular client can have eStream active.	10
1.22	eStream will not allow users to run the same application from multiple clients simultaneously if the license prohibits it.	10
1.23	eStream client user interface will support HELP and ABOUT functions, including links to websites with FAQs and support access information	10
1.24	To start the process of having a particular ASP's application subscriptions known & kept updated on a client, one must visit the associated ASP website with that client at least once.	

3.0 Server Requirements

The following are performance and functional requirements for the server portion of eStream 1.0:

ID	**Description**	**Priority**
1.0	eStream provides user and account management capabilities.	10
1.1	User Account Creation/Deletion supported.	10
1.2	User Account is able to subscribe/unsubscribe to Applications.	10
1.3	User is able to view billing and account information.	10
1.4	User is able to change password/address/billing information online.	10
1.5	User is able to list all available & subscribed	10

	applications.	
1.6	User is able to access online help/doc, including an FAQ database.	10
1.7	Omnishift provides interfaces to facilitate customer support by third parties.	10
1.8	User is able to enter/modify data securely.	10
1.9	Both IE 4.0/later and Netscape Navigator 4.0/later browsers are supported.	9
1.10	ASP agent [i.e., special administrative user at the ASP] is able to access all user information.	10
1.11	ASP agent is able to disable a user.	10
1.12	ASP agent is able to modify license information for a user.	10
1.13	User is able to add additional users to the account.	10
1.14	Web Server is highly scalable.	10
1.15	Servers are able to operate in non-English language.	9
1.16	ASP may operate eStream system with single server.	9
1.17	Flexible access/export of billing information is supported, to facilitate 3rd party billing systems.	10
1.18	eStream server software and eStream apps can be upgraded w/o impacting installed eStream client software. All upgrades are backwards compatible.	10
2.0	The eStream framework [ASLM Server] provides a mechanism to validate the usage of application components with respect to billing models.	10
2.1	ASLM server is able to validate users to use specific applications.	10
2.2	ASLM server records all usage activity down to the granularity necessary to support billing models. The granularity will be reasonably large.	10
2.3	ASLM is able to release license on explicit request or timeout from the client.	10
2.4	ASLM is portable across a wide variety of platforms and operating systems, including but not limited to: Windows NT4, Windows 2000, Solaris UltraSPARC, and Linux.	10
2.5	ASLM servers are fault-tolerant.	10
2.6	ASLM servers are scalable.	10
2.7	ASLM server is able to report Denial of Service attempts.	10
2.8	ASLM server reports illegal accesses.	10
2.9	ASLM is able to register its presence/load to the Web Server(s).	9
3.0	eStream Framework provides management and monitoring tool (EMMT) to manage the servers.	10

3.1	EMMT is able to start/stop servers in the eStream framework.	10
3.2	EMMT is able to monitor server activity for all servers in realtime.	10
3.3	EMMT is able to configure the servers.	10
3.4	EMMT is able to provide historical reporting.	9
3.5	EMMT is able to display information graphically and in spreadsheet format.	8
3.6	EMMT is able to raise alarms on predefined events.	9
4.0	eStream framework provides a mechanism to deploy the application via the eStream Builder.	10
5.0	eStream framework support a variety of licensing models.	10
5.1	Floating license model is supported. n User – k Licenses	10
5.2	Names User License model. (Special case n=k)	10
5.3	Time based licenses at billing granularity.	10
5.4	High water mark license.	10
5.5	Node locked licenses.	8
6.0	App Server is able to Authenticate client's accesses (via AccessTokens) completely locally.	10
6.1	App Server encrypts returned data (via a random key chosen by the client); it must be computationally infeasible to steal an application's code while it is being distributed or to determine which application a client is running.	10
6.2	App Server is as stateless as possible to allow client to switch to alternative app server w/o significant overhead. "Stateless" means that there is no server context that would be lost if the server went down; one classic example of this is that "file open" is recorded on the client, not on the server.	10
6.3	App Server is optimized to respond to requests with minimal server load, thereby maximizing scalability.	10
6.4	App Servers may be grouped along with any number of other such servers into a farm with minimal inter-server interactions (as to maximize [illegible]	9
6.5	App Server communicates with clients thru firewalls.	10
6.6	App Server communicates with clients efficiently (e.g., via persistent HTTP connections).	10
6.7	App Server is able to install new eStream sets w/o having to go down.	10
6.8	App Server is robust, able to run for long periods	10

	without crashes (i.e. no resource leaks, and handles most/all failure modes for system operations); 24/7 operation.	
7.0	App Servers, ASLM Servers, ASP Web Server and EMMT communicate through a database which will include but need not be limited to Microsoft SQLServer.	10

4.0 Builder Requirements

The following are performance and functional requirements for the builder portion of eStream 1.0:

ID	**Description**	**Priority**
1.0	The Builder installation monitor runs in the background, when an eStream application is installed as part of its preparation or building capabilities.	10
1.1	The Builder installation monitor captures all the updates to the System Registry that take place during the install.	10
1.2	The Builder installation monitor records all the files created in the two kinds of directories: the install directory and the common directories.	10
1.3	The Builder must be able to gather initial set of application profile data. This data consists at least of the page access pattern for starting and immediately shutting down an application	10
1.4	The Builder must package the eStream Set into an easily manageable packages suitable for ASP administrators to download to their servers.	10
1.5	The Builder must be able to collect per-user profile data from the Profile Server and merge the profile data into a combined data usable for updating the profile data in the appInstallBlock.	8
1.6	The Builder should be run in an environment where no other applications are running.	10
1.7	The Builder should provide functionality to create installation set(s) for each of the clients eStream 1.0 is going to support.	10
1.8	It should be possible to change the appId of the eStream set when an ASP wants to "install" the eStream set in order to host it.	10
1.9	It should be possible to create a merged eStream set	10

	for a suite of applications.	
1.10	It should be possible to test the eStream Set created by the Builder using a stand-alone tester and not require the eStream client+server programs.	10
1.11	The appInstallBlock should have support for indicating upgrades at the support site	10
1.12	In the process of creating an eStream set it should be possible for the user to delete file entries and registry entries manually to "trim" the eStream set if she so desires assuming the user knows what she is doing.	10
1.13	The Builder should be run in a clean machine with as few software installed/upgraded as possible.	10
1.14	The Builder should support individual applications in a suite even if the installer of the suite doesn't allow installation of individual applications.	10
1.15	The Builder must be able to create an initial set of cache contents for the eStream client and allow the initial size to be selectable by the user or automatically.	10

5.0 Client Use Cases

5.1 USE CASE: Installation of eStream client code

- Obtain eStream client code bits.
- Install z: file system hooks & setup to have z: mounted at appropriate time.
- Install eStream client code, which services z: file sys requests from local cache or from servers & which handles sideband communication w/ servers, and setup to activate estream client code at time desired by user (boot, login, on demand).
- Install NoCluster.sys to disable page fault clustering at system boot.

5.2 USE CASE: Installation of application

- Obtain AppID & App Server name for installation from SLM Server.
- Download AppInstallBlock information.
- Perform initial installation & setup for app, after checking system for previously installed version of app & issuing any appropriate warnings.

5.3 USE CASE: Uninstallation of application

- Remove all registry/DLL/filesys changes associated with app installation.
- Remove all other data associated with application.

5.4 USE CASE: Uninstallation of eStream client code

- Remove z: file system hooks, eStream client code, & nocluster.sys.

5.5 USE CASE: Execution of eStream client code

- Respond to z: file sys requests and detect when new eStream app is referenced.
- Support Client UI requests.

5.6 USE CASE: Execution of application

- Obtain Access Token & list of App Servers from SLM Server.
- Contact App Server(s) as desired to obtain file system data.
- Respond to running application's requests, collect usage data. Cache portions of application, file system info, & user preference info.
- Detect server connection issues (apparent loss of connection or connection response below acceptable threshold) & licensing issues; negotiate with ASLM Server as needed.

6.0 Server Use Cases

6.1 USE CASE: Create an Account

- Customer brings up browser and connects to ASP Web Server
- Screen display shows "create account", customer selects and enters required account info (billing info, owner userid, pword, etc)
- ASP Web server writes account info to Acct DB using **AddAccount** where a unique Account ID is assigned
- Account ID is returned via web page

6.2 USE CASE: Create a User

- Customer brings up browser and connects to ASP Web Server
- Customer enters their userid and pword
- ASP Web server contacts Acct DB, using **AddUser** userid and initial password, gets Acct info and displays to Customer
- Customer selects "add user" and enters required user info (username, address, email etc)
- ASP Web server writes user info to Acct DB updating account info

6.3 USE CASE: Modify Account

(includes disabling an account or user, removing users from accounts, changing pwords etc)

- Customer brings up browser and connects to ASP Web Server
- Customer enters their userid and pword
- ASP Web server contacts Acct DB, passes along userid and pword, gets Acct info and displays to Customer
- Customer selects "update info" and enters desired changes
- ASP Web server writes updated account info to Acct DB

6.4 USE CASE: AddSubscription

- Connect to ASP web server
- Enter account number, username, password
- Verify that user is account admin using **GetUserPermissions**
- Get list of possible subscriptions (using **ListPossibleSubscriptions)**
- Get list of current subscriptions for account (using **ListCurrentSubscriptions)**
- Display in page – User chooses a subscription and license type
- Display a screen to allow the user to configure the license. For a floating license, allow selection of users, etc.
- Call **CreateSubscription** to compose the new subscription for each user and create licenses.

6.5 USE CASE: Building an eStream set:

- Start w/app **CD-ROM**, and a freshly installed OS (plus latest service pack?).
- Install app into Z: drive (could just be a regular network drive)

- A **special system monitor** logs all registry changes and file system changes during the install.
- File system changes to C: during install probably need to be spoofed (or have a registry entry point to Z: instead), especially newly added directories, so need to do the appropriate thing.
- From this log and the actual files as installed on the machine, the **eStream set builder** creates the eStream set, which is a small set of related files.
- Separately, we need to actually set up the app for eStreaming, then run it and collect profile data to seed the initial page prediction map.

The **AppServer UI** (interface to user to control an Application Server on a particular machine) presents the following management functions:

A. Starting a server:

- **AppServer UI** always indicates whether an AppServer process is up and running (and alive w/status), and if present prompts for restarting the current server process.
- Otherwise it goes ahead and starts up the AppServer process and reports any errors.

B. Stopping a server:

- Simple, just stops any running servers, gracefully, perhaps prompting user for ungraceful shutdown if not successful.

C. Install eStream set:

- Each server is configured with a specific eStream set directory, under which it places (in their own individual directories) the actual eStream set contents (a few files on the native file system).
- User indicates to **AppServer UI** where to find the eStream set package provided by Omnishift. **AppServer UI** authenticates the package, and verifies its integrity, and if successful, unpacks and places the constituent files in the server's eStream set directory.
- Note that it is possible for the eStream set directory to live on a file server shared by other **Application Server** machines, so installation may be required only once ([illegible] happen once per machine, and a separate [illegible] is responsible for replicating eStream sets across a farm to ensure the farm machines are symmetric).
- How does the server know a new eStream set is available? Each set is assigned a VolumeID, and the set contents can be placed under a directory with the same name as the VolumeID. The install is synchronized via a AppList file, which just lists the valid VolumeIDs, which the **Application Server** only reads, so an entry is added at the end of the install procedure. The **AppServer UI** then must send

some kind of message/signal to the Application Server to have it resync with the file (and start serving the new app).

- Also note that eStream set install is doable without bringing down the server (or any server in the farm.
- Having done this, it will probably be necessary to notify the **SLM** and **Account Servers** that a new app is available. With some scripts provided by omnishift, this sould be done by a human administrator. They need to know the VolumeID of the app that was installed along with the full name, so that the client can initiate an app install procedure via the VolumeID (the server can then provide the **AppInstallBlock** which probably has a fixed reserved global FileID).
- Questions: What if app is already installed (want to allow reinstall or force remove first)? What if app is being upgraded (probably also should be a remove and then install)?

D. Remove eStream set:

- First we probably have to disable the application on the **SLM/Account servers**.
- This probably will require sending some kind of message to the Application Server (if running) to stop serving the given eStream set, and then waiting for any active connections to expire.
- Then we can just remove the entry in the AppList file, and delete the file system image.

E. Configure Application Server:

- The **AppServer UI** presents various configuration options to the user (stuff like logging, port #, threads, etc.) Some may require restarting the **Application Server** to take effect, others may take effect immediately.

Another activity that occurs, automatically, is the processing of profile data. It is not clear what the page prediction map looks like, but clients will periodically send profile data to the **Application Server**, which aggregates it, and must store it persistently, and to allow new clients to benefit from improved prediction. There may need to be a special module that can take the aggregated profile data to modify the prediction map.

5.6 USE CASE: Acquire Access Token

Where are we?

- Client PC has installed the subscribed app and has received a subscription token, and the name/IP of SLM Server.
- Customer is accessing an app file and doesn't have an access token for it, yet. (i.e. double clicking z:\word.exe).

Players involved: client – cache mgr, SLM Server and indirectly, User/Account/Sub/Rights DB.

What happens:

- Client contacts the SLM server and gives: subscription token, user/passwd.
- SLM server looks into the user/acc/sub/right DB to
    - Authenticates user and password; may return: "invalid user".
    - Authenticates subscription token; may return: "invalid subscription token"
    - Look at the Accounts container and see if any licenses are available. If so, check it out by creating a new access token and updating the accounts container. It may return: "can't get license".
- Return an access token to the client and a list of app servers.

6.7 USE CASE: Process File Request – steady state

Where are we?

- Client has installed the app and has a list of app servers,
- Client is holding a valid access token that it acquired from the SLM server.
- Client, while processing an IRP, needs to access portion of file on the app server.

Players involved: client – cache mgr, app server. NO SLM server or no user/account/dub/rights DB.

What happens:

- Client contacts one of the app servers and gives: access token, App ID, File ID, length and file offset.
- App server quickly verifies the expiration date on the access token.
    - It *must not* need to contact the user/account/sub/rights DB to do this. It only cares about the time-validity of the token. If token has expired, return some kind of an error back to the client.
- App server locates the data and sends it back to the client.

NOTE: we are simplifying this quite a bit when discussing the scenarios because we are not sure exactly how we are going to

manage the server farms. Another key question is MUST all app servers host all estream apps ?

6.8 USE CASE: Renew an Access Token – steady state

Where are we?

- Client acquired an access token from the SLM server.
- While running the app, client sees the needs to renew the access token. This may happen synchronously when the user touches one of the app files, or by a timer-driven client daemon that periodically renews an access tokens before it expires.

Players involved: client – license manager, SLM manager, and indirectly, user/accounts/sub/right/ DB.

What happens:

- Client sends an access token to the SLM server.
- Check the time-validity of the access token.
  Assumption: SLM server assumes that only valid access tokens can be renewed. An expired token implies a lack of renewal, which implies releasing the license. SLM server can try to acquire the license, but there is no guarantee that it will succeed.
    - If token is expired app, goto Scenario: Acquire Access Token.
- SLM server accesses the user/account/sub/rights DB to:
    - Generate a new token that will expire some time in the future (configurable parameter).
    - Update the account container in user/account/sub/rights DB.

Return the new access token.

6.9 USE CASE: Validate user request for access to an application server

Procedure:
Receive username, password, machineNodeID, subID and appID from the Client
Query AccountDB for license to access application appID in subscription subID
If (no valid license) then
  Send FailureReason to Client
Else
  Send accessToken, appServers to Client

6.10 USE CASE: Add subscribable application from an account

Interface Required:
SLMServer::AddSubscribedApp(accountID, subID)
Procedure:
Receive accountID, and subID from the Client
Check for valid accountID, and subID on AccountDB
If (no valid accountID or subID) then
Send FailureReason to Client
Else if (subID is not already subscribed under accountID)
Add Subscription subID to Account accountID in AccountDB
Send Success to Client

6.11 USE CASE: Remove subscribable application from an account

Interface Required:
SLMServer::RemoveSubscribedApp(accountID, subID)
Procedure:
Receive accountID, and subID from the Client
Check for valid accountID, and subID on AccountDB
If (exist subID in accountID) then
Remove Subscription subID from Account accountID in AccountDB
Send Success to Client
Endif
Send FailureReason to Client

6.12 USE CASE: Monitor/management tools

Interface Required:
SLMServer::GetTrafficHistory()
SLMServer::GetUsageInfo(userID, appID, subID, accountID)
SLMServer::GetCurrentTraffic()
SLMServer::AddServer(serverID)
SLMServer::RemoveServer(serverID)
SLMServer::RemoveClient(userID, serverID)
SLMServer::GetErrors()
SLMServer::DumpErrors(filename)
SLMServer::DeleteErrors()
AppServer::GetTrafficHistory()
AppServer::GetCurrentTraffic()
AppServer::GetErrors()

Procedure:
SLM Servers keep track of traffic info. The monitor/management tool can query the SLM/App Servers anywhere for traffic info. Some examples of traffic data:

- Traffic history of particular server on number of clients served per unit time
- Monitor length time a userID used application appID under subscription subID and charged to accountID
- Monitor current load information on all servers (SLM server and app server)
- Allow admin manually add/remove some servers from the pool.
- Allow admin to kick some clients off the server.

The monitor/management tool can also be used to display a list of errors logged by the servers.

- Monitor errors and be able to categorize by error type
- Monitor errors occurring between certain time periods
- Monitor errors reported by a particular server
- Manage errors to dump the errors to a file
- Manage errors and delete a subset of errors

Finally, the monitor/management tool can check for any illegal accesses.

- Monitor failed attempts to access SLM Server with bad password, especially on repeated failed attempts in a short time frame.
- Monitor any attempts to use a particular license and failed.
- Monitor access to SLM Server from non-typical IP addresses for a particular account. The server is required to save the history of IP addresses of accesses to a particular subscription account.

6.13 USE CASE: Adding a new application server.

Summary:
An application server's functionality is to provide applications eStream sets to client application. An application server is generally added to the system to provide greater scalability and/or to provide additional application support.

Actors:

1. ASP administrator: Responsible for installing the server and the applications.
2. ASLM Server(s): The ASLM server needs to be notified of the presence of an additional application server and the services it provides.

Inputs:

1. Application server(AS) installer
2. Application eStream sets. These may be available from one of the following location: AS installer, some other AS or Farm Manager Server(some central repository) .

3. SLM Server location(This input may not be required based on scalability solution that we decide on).

Processing:

1. Using the AS installer install the application server.
    a. <Server install use case to be added here? Later>
2. Copy the Application eStream sets. There are several options here:
    a. Provide the eStream sets as a part of the installer.
    b. Provide a script to ftp to another Application server and copy the eStream sets.
    c. Provide a management tool to manage the copying of the eStream sets. From the ASP's perspective this is the best solution. A tool which provides tracks the application would be useful to manage the load.
3. Configure the server. The server needs to know the additional application sets that it supports(? This may not be required).
4. Start the server.
5. Register the server with other SLM servers. The following options apply:
    a. Multi-cast the "new server and services" message to the SLM servers.
    b. Register the server to a local object server which in turn notifies the object servers across the system. CORBA model supports this.
    c. Using the resonate model(described below), all appservers are essentially the same server. ie Address app.foo.com will point to a set of app servers. A new server enabled will resonate software will automatically register itself with the resonate scheduler. (How do we make the resonate scheduler aware of the applications available on the app servers?)

Outputs:

1. The App server is installed and running with a set of applications available on it.

6.14 USE CASE: Removing an Application Server.

Summary:

An ASP administrator may decide to remove an application server from the system for various reasons. Removal of server from the system would result in notification to the rest of the SLM servers that it is no longer available for servicing the objects.

Actors:

1. ASP administrator.
2. SLM Servers.

Inputs:

1. Application server running on the machine.
2. ASLM Server(s): The ASLM server needs to be notified of the presence of an additional application server and the services it provides.

Processing:

1. Stop the application server. This will result in the Application server informing the rest of the ASLM servers that it will no longer take any requests. This in turn may result in an application being unavailable for usage. Depending in the framework used, this can be done in one of the following ways:
    a. Multi-cast the message to the ASLM servers.
    b. Just stop the server in the CORBA framework. The local ORB server will notify the unavailability of the resource to the rest of the framework.
    c. Using the resonate model to scale would imply that you just stop the server the resonate agent on the server will notify the resonate scheduler to deregister the servers. (However its not clear if you can also deregister the objects served by the server.).

Outputs:
1. ASLM servers are notified of the removal of the resource.

6.15 USE CASE: Add a new ASLM server.

Summary:
The ASP provider may decide to add an additional ASLM server to enhance the performance of the system. The additional ASLM server added to the system should be accessible to the ASP's Web Server so that it can direct the clients to the SLM server. (This may not be required if we deploy the Resonate model of scaling).

Actors:
1. The ASP administrator.
2. The ASP Web server.

Inputs:
1. ASLM installer
2. Web Server location(This input may not be required based on scalability solution that we decide on).

Processing:
1. Using the ASLM installer install the application server.
   <Server install use case to be added here? Later>
2. Start the server.
3. Register the server with ASP Web Servers. The following options apply:
    a. Multi-cast the "new server and services" message to the Web servers.
    b. Register the server to a local object server which in turn notifies the object servers across the system. CORBA model supports this.
    c. Using the resonate model(described below), all ASLM are essentially the same server. ie Address aslm.foo.com will point to a set of ASLM servers. A new server enabled will resonate software will automatically register itself with the resonate scheduler.

Outputs:
The ASLM server up and running.

7.0 Builder Use Cases

7.1 USE CASE: Install Monitoring

- Query builder for CD media and installation executable(s)
- Monitor various registry and file updated during installation
- Merge installation data for all applications in a suite
- Relocate files from C: to Z: directory
- Create appInstallBlock and package the appInstallBlock with the application files

7.2 USE CASE: Profiling

- Query builder for application executable(s)
- Monitor sequences of file accesses from OS to the file system as profile data
- Identify the subset of the profile data as the initial cache contents
- Merge profile data and initial cache contents into the corresponding appInstallBlock

8.0 Key Applications for eStream 1.0

Winstone99:
Business: QuattroPro, WordPerfect, Lotus® 1-2-3, Word Pro,
Access, Excel, PowerPoint, Word
High-end: Adobe® Photoshop, Adobe® Premiere, Microsoft® FrontPage,
Sonic Foundry® Sound Forge
Content Creation Winstone 2000: Macromedia Director, MacromediaDreamweaver

Please note that release of Business Winstone 2000, which was originally slated for 6/27/2000, has now been postponed until the Fall Comdex & will be called Winstone 2001. As soon as the contents of this suite are released, we should move quickly to assess our support for its application set.

eStream 1.0 High Level Design

Version 1.0

[redacted]

Introduction

This document describes the high level design for the eStream 1.0 product. It is essentially a summary and a tying together of the low level designs for each component in the system. The organization of this document is:

- Basic overview of the entire system
- Block diagrams for the client, server, and builder portions, showing all major components
- General discussion of each component, and pointers to the low level documents for these components
- A list of known issues

To understand the problem being solved in this design, see the "eStream Requirements Document" for information.

Note that this design is for a Windows NT4.0 and Windows 2000 client **only**. As work progresses on a Windows 95/98 client, the designs here will be updated.

Overview

eStream 1.0 encompasses the following basic features:

1. A distributed file system for application files, residing on a server and cached on a client.
2. A small client "player" program to allow local execution of applications that reside on the servers.
3. Authentication using tokens supplied by a license server to each active client.
4. A managed database of information about applications available to client machines, and subscription and usage data for each registered user.
5. Integration with service provider web servers to allow users to subscribe to apps and manage their accounts.
6. Monitoring of all servers to detect problems and allow automatic failover.
7. A build system that analyzes applications and enables them to be executed by the client and server.
8. Anti-piracy features to discourage unauthorized copying and use of subscribed applications.

As a way of overview, here are the processes that take place to enable and execute a Windows application from a client machine.

- The eStream builder is used to create an *eStream set* for the application. The application is installed on a clean machine, with the builder tools running. These will monitor all file installs and registry updates required to run the application, and encode them into a binary file—the eStreamSet—that will be installed on a service provider's eStream application server (app server).
- A user must download and install the eStream client (ECE) onto her machine, and register as a valid user from a service provider; this will be done using the service provider's web site.
- The user will subscribe to an application from the service provider; a browser module on the client machine will be notified and send a message to the ECE about this event.
- The ECE will communicate with the service provider's eStream license server (Slim server) to verify the newly subscribed app and all permissions, and will install a small portion of the application onto the client system—essentially, the registry entries, shortcuts, and small shared files necessary for execution.
- All application files that are not installed on the client will be accessed via a separate eStream file system (EFSD)
- The user will now see standard shortcuts for subscribed applications, exactly as though the app were installed locally.
- Starting an application, via a command line or double-clicking a shortcut, will cause the client machine to start executing the application on the EFSD. This means the virtual memory manager will request pages from the EFSD during page faults.
- These requests will be forwarded from the EFSD to the eStream cache manager (ECM), a component of the ECE, and on to the app server, assuming the page requested is not in the cache.
- Before any page request is fulfilled by the ECE, the client license subscription manager (LSM) will check that the user has permission to run the application, requesting an *access token* from the Slim server if an existing one has expired.
- This valid access token is sent from the client to the app server for every page request; this authenticates the request.
- The server monitor will be continually checking the state of the app servers and Slim servers. If any are down, it will take them off line.
- The client has a list of valid Slim and app servers for each registered service provider and subscribed application. If response time for any of these is bad, it will stop using it and fall back on the rest.

Block diagrams

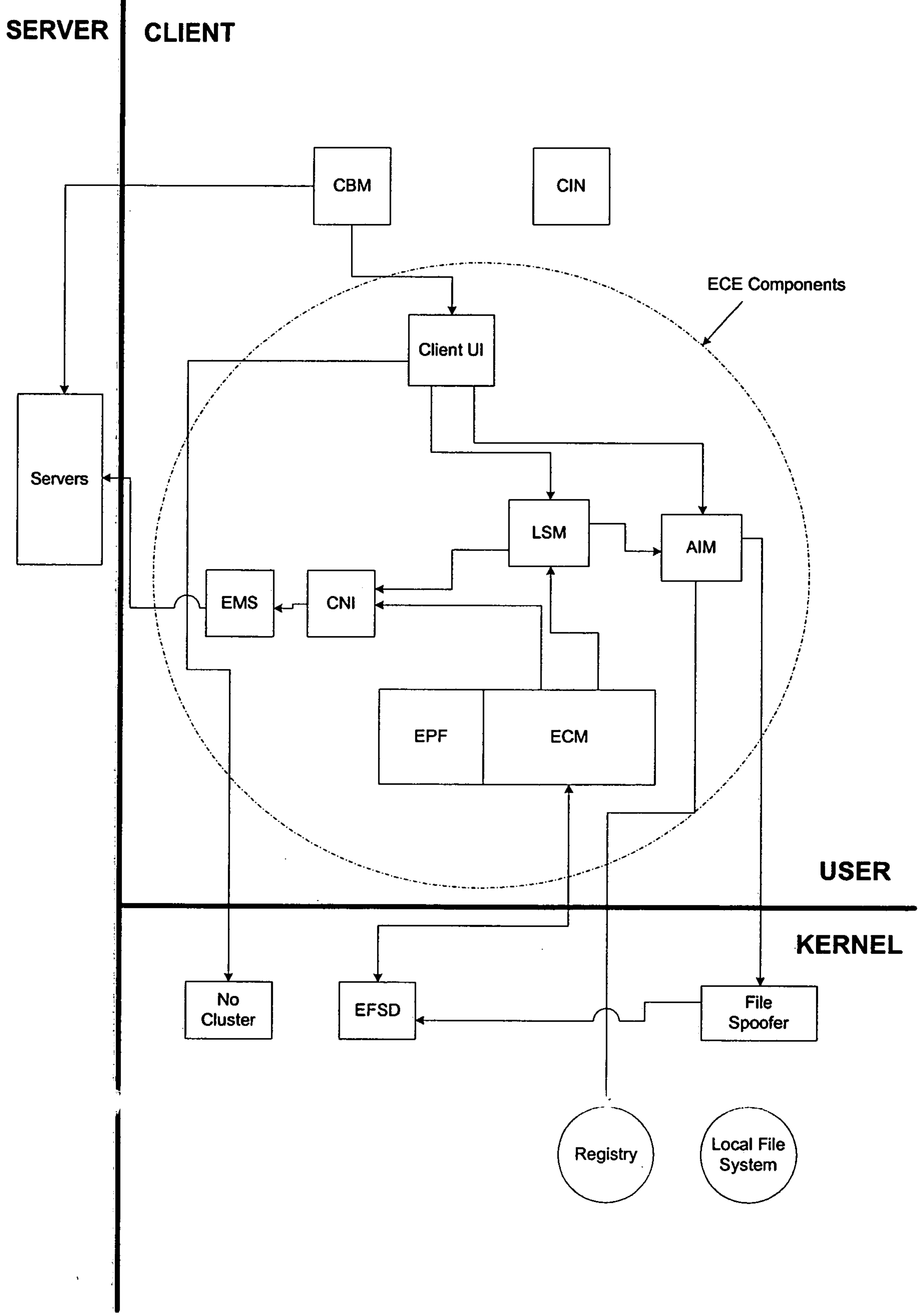
The following are simple block diagrams of the client and server components. Some conventions:

- A box represents a **logical eStream component**. A component may exist as a distinct process, or it may be grouped with other components into a common process.

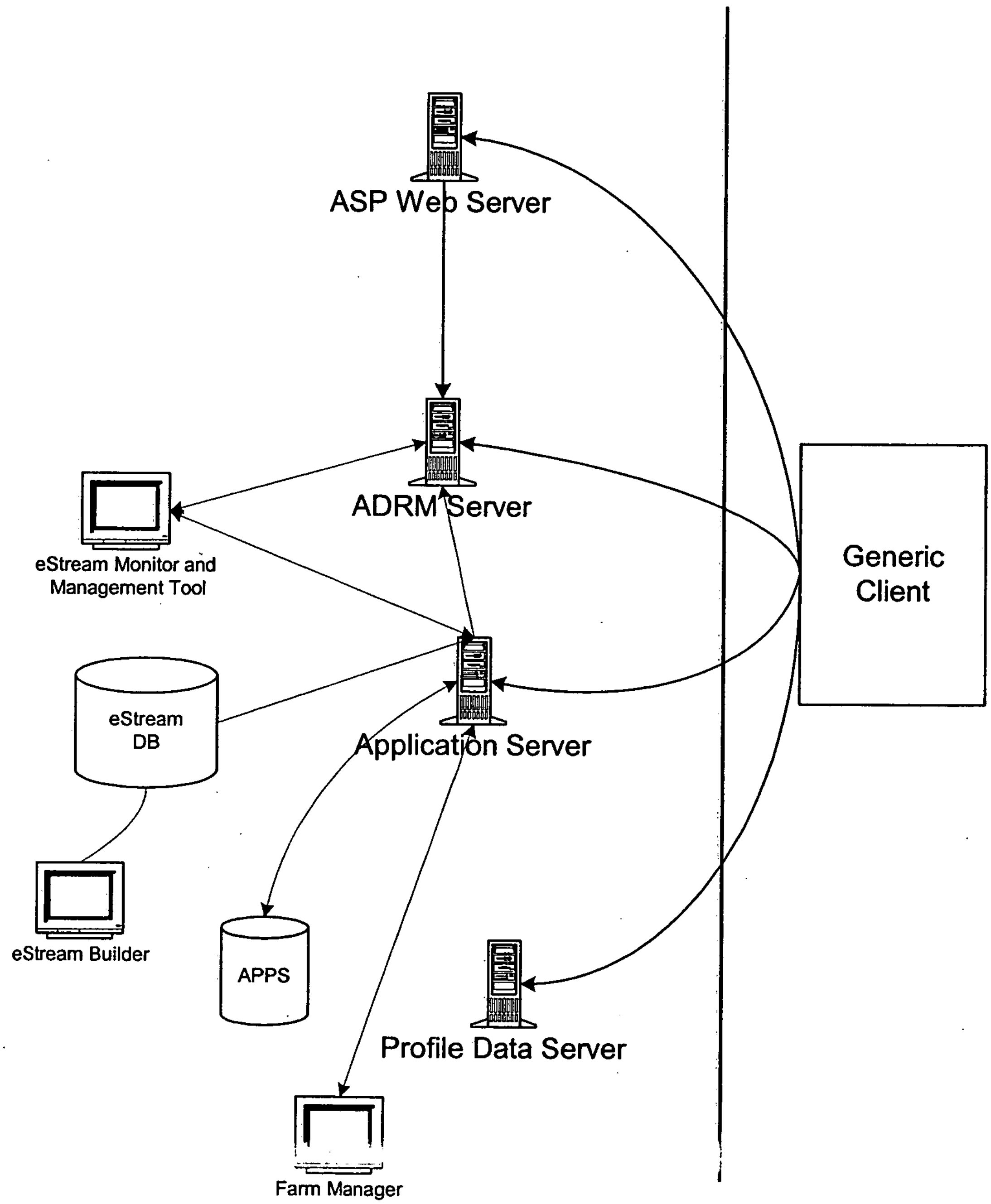
- A line between components represents an interface call from one to another. If A calls B, there's a arrow on the end of the line at B. If A and B call each other, there's an arrow on both ends of the line.

Note that data stores are **not** represented in these diagrams; if a data store is centrally managed, then there is a component that has interfaces to allow access to these data.

eStream Client Block Diagram



eStream Server Block Diagram



eStream Builder Block Diagram

???

Component descriptions

Client components

The eStream client consists of the following components illustrated in the diagram above:

- ECE: the eStream Client Executable. This is the aggregation of several user space components into a single executable, operating as a Windows service.
- LSM: the License Subscription Manager (part of the ECE). This tracks and handles all required user information needed by the client: service providers, subscriptions, and access rights.
- AIM: the App Install Manager (part of the ECE). This is responsible for installing all necessary bits onto a client machine in order to run a subscribed application. It also uninstalls all local app bits when unsubscribing.
- ECM: the eStream cache manager (part of the ECE). This is the user-space component that handles requests from the EFSD, and manages the on-disk and in-memory cache of file contents.
- EPF: the eStream PreFetch component (part of the ECE). This works closely with the ECM to handle prefetches of pages for running eStream applications (as opposed to demand fetches, handled by the ECM).
- CNI: the Client Network Interface (part of the ECE). This manages queues of requests from various client components to the app and Slim servers.
- EMS: the eStream Message Service (part of the ECE). This library, used in both the client and servers, handles the actual network sends and receives between remote machines.
- CBM: the Client Browser Module. This is a client-side web browser plugin that is used to handle notification from a service provider's web server to the ECE, when user updates have taken place.
- CIN: the Client Installer module. This small component installs, upgrades, and uninstalls all the required client software.
- FSP: the File Spoofer. This is a kernel-mode driver that is used to redirect requests, intended for local filesystems, to the EFSD. It is a file system filter driver that sniffs all Create requests to the necessary local FSDs, compares the filenames with a list of files that must be spoofed, and if a match is seen, redirects the request to the EFSD.
- EFSD: the eStream File System Driver. This is a standard Windows NT FSD, handling all necessary FS requests from the I/O Manager. It ultimately sends these requests to the ECM to be satisfied (either locally or remotely).
- NoCluster. This is a kernel-mode driver that simply disables the system page clustering for threads running as part of eStreamed applications.

ECE

The ECE is the Windows service that comprises the bulk of the user-space eStream client software. It provides an overall main program loop, as well as the user interface component for all client components that must communicate with a user.

LSM

The LSM tracks current subscription information and determines the need for license validation. It is informed of subscription changes from the client UI, and is queried by the ECM to validate accessibility to different applications, based on the license model for the subscription to that application.

The LSM has a few major tasks:

1. Keep track of what subscriptions the current user has available from all ASPs
2. Determine which application a given file is a part of
3. Acquire an access token to validate a license for file requests that require one

There are two ways that the LSM updates its list of known subscribed applications:

1. It may be informed of new subscriptions, or of applications that are unsubscribed, by the client UI, as part of a browser plugin in conjunction with an ASPs web site.
2. It may asynchronously poll an ASPs Slim servers to get updated lists of subscribed apps.

AIM

The AIM is the contact point for installation and uninstallation of applications on a client machine. It gets the requests from the LSM to install applications when the user subscribes to them, and it gets requests from the Client UI to uninstall applications.

The AIM manages application installs on the client machine. It keeps track of what applications have been installed on the client machines, where they have been installed and the various components that are part of the installation. It contacts the application servers to get the AppInstallBlock. The AIM uses the AppInstallBlock to then make the appropriate calls to the file spoofer; to install some files on the local disk; to "warm" the cache and to update the start menu and other short cuts as needed.

ECM

The ECM is part of the ECE. Its goal is to:

- Handle all file requests from the EFSD, either by using previously cached contents or requesting the contents from a server.

- Work with the LSM to insure that all applications have appropriately validated licenses before their files are accessed.

The ECM handles the volatile and non-volatile eStream cache on the client machine. It performs demand fetching from the appropriate server(s). Based on the client's observed behavior, it compiles updated profiling data, which may periodically be uploaded to a server.

EPF

The ECM is part of the ECE. Its goal is to intelligently use prefetching of file data to reduce latency of pages requested from the EFSD; this prefetching may result from profiling data or heuristics.

CNI

The client network component is the common point of connection between the rest of the eStream client components and the various eStream servers. Any client module that calls an interface of a server does so through the network component.

EMS

CBM

CIN

The client installer is a simple InstallShield (or simpler) application that will install all of the required client software.

FSP

The purpose of the file spoofer is to redirect file system accesses from some non-eStream drive. This may be necessary in order to support applications running under eStream that are hard-wired to access files in a specific location. The file spoofer may also be used if we are interested in providing a version of some system file different from the one actually on the client machine.

The file spoofer will intercept File Create calls for files that we are interested in spoofing and ensure that these creates are redirected to a file we specify. The redirection could be to a file on the EFSD, or to another, non-eStream'ed file.

EFSD

The EFSD provides standard kernel file system interfaces to the I/O manager and other kernel-mode components. It works with the NT Cache Manager to efficiently cache file

and directory contents. Its view of the ECM is essentially like that of a disk driver, sending primarily read and write requests as needed.

No Cluster

The VM clustering disabling driver (aka NoCluster) disables virtual memory clustering under Windows. While we don't fully understand all the implications, using this driver substantially reduces the average file system paging request size and can dramatically improve performance of eStream, especially on slower connections.

Virtual memory clustering, as implemented in Windows NT/2000, is intended to improve performance when paging to and from physical disks. If possible, we would like to disable clustering only for those threads/processes that will be doing a significant amount of I/O to the eStream file system.

Server components

The following are the server components for eStream 1.0:

- App Server. This is essentially a file server for eStream sets. It satisfies requests for pages from eStream files from the client.
- Slim Server. This handles requests from a client for user and service provider information, and grants access tokens to the client for executing eStream applications.
- Web Server. ???
- Monitor. This enables an administrator to view the server components. It regularly pings the various servers, takes disabled ones off line, and adds new ones to the pools.
- eStream Database. This tracks all user information and server resources for a given service provider.

App server

The application server is there to handle read requests for files accessed by eStream clients. Any file accessed on a client through the EFS can have this read request passed to an app server.

This will be the hardest working eStream server. It will respond to both synchronous (demand fetching) and asynchronous (prefetching) page requests from many different clients, for many different types of applications and files within those applications.

Slim server

The Software License Management (Slim) server is responsible for:

- Managing data related to users, the groups they belong to, and the applications they are subscribed to
- Validating the licenses for applications executing on clients
- Tracking all outstanding licenses currently in use

ASP web server

This describes only those interfaces on an ASP web server that relate to handling eStreamed applications.

Logically, the ASP web server is the backend web interface for user requests—e.g., get billing information, subscribe to a new app, or request a list of all possible apps a user can subscribe to. In the current model, the web server doesn't actually handle these requests, but instead passes them on to the appropriate eStream-centric server.

Monitor

The monitor utility is responsible for monitoring the overall health of the system. It is responsible to report server status, server traffic, illegal access etc. It will ping the Application Server and the Slim servers to gather the statistics and display them.

Database

Builder components

These are the builder components for eStream 1.0:

- ???

Estream 1.0 Planning Document

[redacted]

Low-Level Design Status/Plan

Sub Components	Owner	LLD Design Doc completed	LLD review Complted	Estimates for Impl	Impl and Unit Test Completed
Content					
Install Monitor	Sanjay	Done	Done	3 wk	
Builder GUI	Sanjay	Done	Done	1 wk	
FSRFD (Drivers)	Sanjay	Done	Done	2 wk	
AppInstallBlk structure	David	Done	Not needed		
Profiler	David	Done	Done	2 wk	
File Access Monitor	David	Done	Done	1 wk	
Packager	David	Done	Done	1 wk	
eStream distribution	**Bob**	[redacted]	**Status TBD**	**TBD**	
Server Group					
Web Server	Bhaven	Done	Done	8 wk	
Monitor	Mike	Done	Done	4 wk	
SLiM Server	Amit	Done	Done	2 wk	
App Server	Sameer	Done	Done	4 wk	
Admin UI	Bhaven	TBD	TBD	TBD	
End User UI	Bhaven	TBD	TBD	TBD	
Common Server Components	Mike	Done	Done	3 wk	
Messaging	Sameer	Done	Done	3 wk	
Threads Package	Sameer	No Document		1 wk	
Security Design	**Igor/Amit**	**Not Done**	**Not Done**	**TBD**	
Client Group					
Cache Prefetching	Anne	Done	Done	1 wk	[redacted]
LSM + Plug in	Anne	Done	Done	1 wk	
Client UI	Anne	Done	Done	1 wk	
Client Installer	Anne	Done	Done	1 wk	
Start Client	Anne	Done	Done	1 wk	
Application Install Mgr	Nick	Done	Done	TBD	
Piracy	Nick	Done	Done	TBD	
File Spoofer	Curt	Done	Done	1 wk	
eStream File System	Curt	Done	Done	8 wk	
NoCluster Driver	**Curt**	**Status TBD**	**Starus TBD**	**2 days**	
eStream Cache Manager	Dan	Done	Done	8 wk	
Client Network Interface	Dan	Done	Done	2 wk	

Implementation Plan

Milestones

[redacted]

ECM (RAM disk cache) and EFSD executes a local "himom" executable
Photoshop is installed locally and successfully executed from estream sets and appInstallblk produced by builder
App Server and EMS integrated to copy "himom" executable using a dummy client
App Server, EMS and CNI integrated to copy "himom" executable from "himom" estream sets
office is installed locally and successfully executed from estream sets and appInstallblk produced by builder
App Server, EMS, ENI, ECM and EFSD integrated to run "himom" from estream sets on server
Following applications built and tested with local installation

Adobe Premier
Macromedia Director and Shockwave
Corel Suite
Lotus Suite

[redacted] **Photoshop is installed by AIM and executed from estream sets on App server**

No Subscription
No License Management
RAM cache for ECM
Installation of Photoshop using AIM

[redacted] **Photoshop is installed by AIM and executed from estream sets on App server**

No Slim Server
Disk based cache for ECM

Estream includes initial prefetched pages and these pages are prefetched during installation
Fully functional estream bits (includes initial prefetched pages)
Client software is run as a service
App Server is started by Monitor
Admin UI to stop and start app Server
Application subscription from web server
installation on client after subscription

[redacted] Testing environment is setup (configuration of 3 servers and one client)
Photoshop runs with the following additional functionality

Leads for milestone: Amit and Nick
Slim Server
http protocol
CNI supports uique message ids for NAD
Fully functional LSM
Real Accesstokens
Uninstall applications
Anti-Piracy support
AppServer and SlimServer fail-over
File spoofing

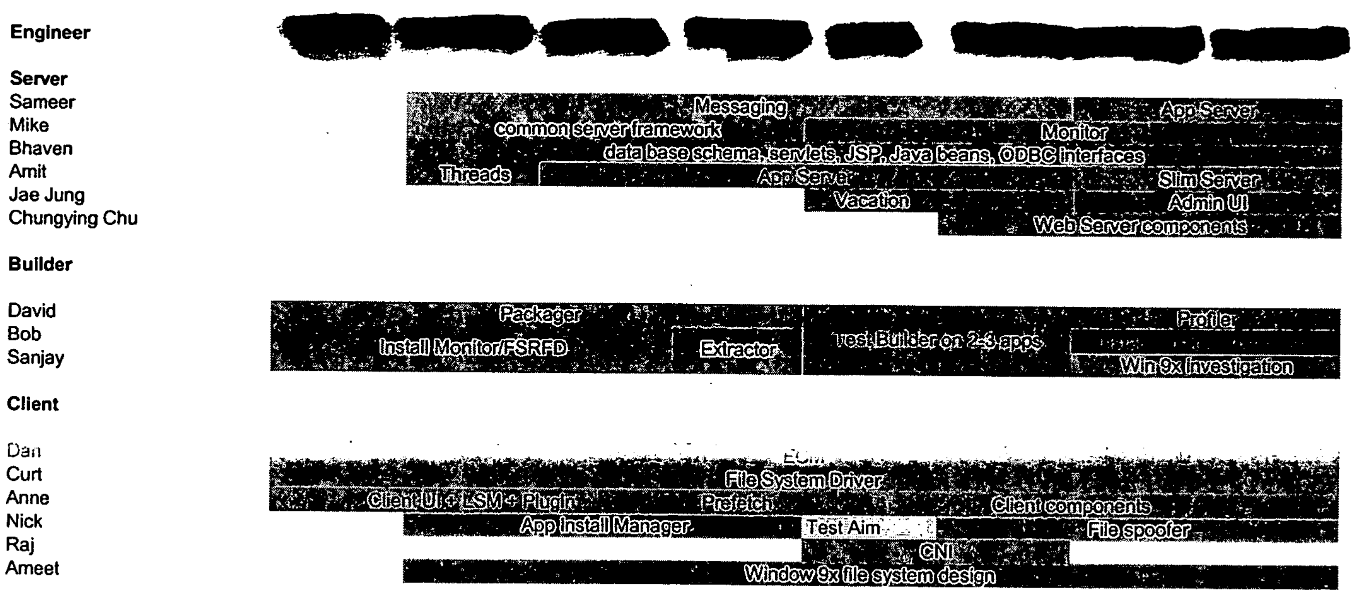
[redacted] Clean builds by integration (George) (Raj will drive this)

[redacted] Office is running with full functionality

Restruturing of client so it can be started at boot time
Performance tuning
Improve robustness
application upgrade
Crash resiliency
All software purified and memory leaks eliminated
(May be) Applets for monitoring server components

[redacted] **Office is removed from desktop of at least one person and reinstalled using estream**

[redacted] Code Freeze



eStream Server Component Framework
Low Level Design

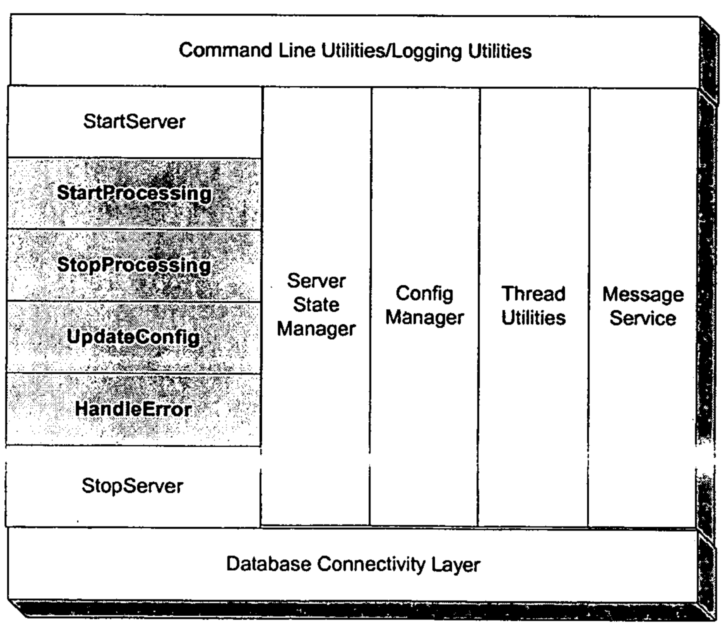
Michael Beckmann
[redacted]

Functionality

The *Server Component Framework* provides a common basis on which server components are implemented. The framework provides a number of services such as common server initialization and configuration, messaging, state management, logging, and error handling. The component framework ties together many of the core utilities provided for the server components.

The advantage of the framework is that heterogeneous server components can be managed in a consistent manner with the expectation that all server components will communicate and behave consistently within the system.

All server components with the exception of the web server will be built on top the *Server Component Framework.* To make use of the *Server Component Framework,* a specialized server component will need to extend the framework by implementing the methods high-lighted in gray. Implementing these interfaces makes the specialized server component "plug-able" within the framework.



The following table give a brief description of each of the routines that need to be specialized by each server component to make it plug-able into the Server Framework:

StartProcessing	Specialized server component routine to request the server component to start processing work.
StopProcessing	Specialized routine to request the server component stop processing work and transition into an idle state
UpdateConfig	Specialized routine to dynamically update configurations while a component is either in the processing or idle state.
HandleError	Specialized routine to handle the occurrence of an error

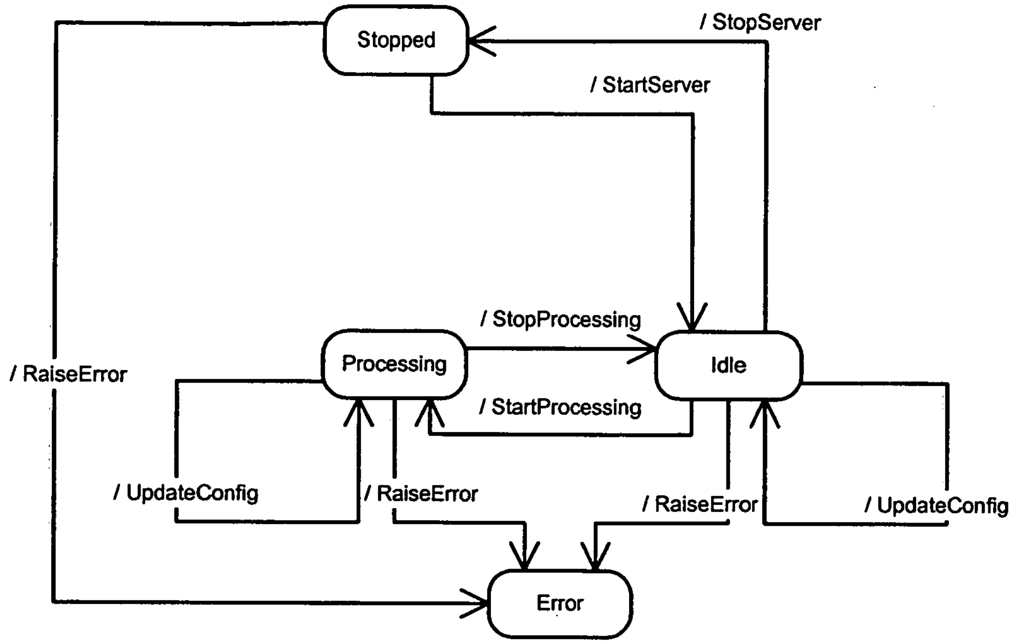
Server State Manager:

At the heart of the server component framework is the *Server State Manager*. The server state manager is a set of interfaces that initiate and manage state changes within a server component. All Server components, by virtue of being built on top of the component framework, can be managed uniformly across a deployment.

The *Server State Manager* implements a simple state machine that is shared between components. It manages the state transitions within the server component. Additionally, the state manager maintains current state information for each server component and logs state transition history in the event that a server component terminates unexpectedly.

As specified above, each server component is required to implement a number of transition methods, with pre-defined signatures, which the state manager will execute when making a state transition.

The following diagram shows the state diagram and the associated transitions:



Message Service:

The *Server Component Framework* depends on a message service which is used by the Server State Manager and Configuration Manager to communicate with the System Monitor.

The *Server State Manager* uses the messaging service to listen for state change requests from the System Monitor which it satisfies by returning the current state, any up-to-date status, and load information.

The *Configuration Manager* uses the message service to request configuration information from the *System Monitor*. Although each server component could easily go to the database for configuration information, it has been decided to go through the monitor as to save db licensing costs.

See below for more details on messaging protocols for the *Server State Manager* and the *Configuration Manager*. Also, refer to the low-level design document for details on the design of the eStream Messaging Service (EMS).

Configuration Management:

The configuration management utility is used by all server components to manage the server configurations. It provides the following functionality:

- Configuration for a server consists of a set of name – value tuples where the values themselves can be a set of name-value tuple.
- Servers can load the complete configuration from the database (indirectly).

- Servers can load the configuration for a given name.
- Servers can load the configuration from a flat file also.

On the Server Manger interface, configuration will appear as a table containing name – value tuples. The table may be hierarchical to represent nested structures containing the values which can themselves be name values. An example of a simple name-value pair would be:

port 8080

An example of nested name values would be:

Applications:
word.exe windows2000sp3
excel.exe win98sp4

On a flat file the configurations will always be name-value pairs. To represent one level nested structure the format would be:

Applications word.exe windows2000sp3
Applications excel.exe win98sp4

A common set of configurable parameters is defined for all server components. These configurations are maintained by the *Server Component Framework* in collaboration with the *Configuration Manager.* All configuration information is persistently stored within the database. The common configurations are used to initialize the server component after the component process has been launched. Refer to the configuration table below for more details on common configurations. Specialized server components can support additional configurations (non-common) depending on the server type. These configurations are read from the database and updated when a server component starts processing. They can also be updated dynamically while a server component is processing through the use of the **UpdateConfig** interface.

The list of common configurations include:

Information	**Supports Dynamic Config**	**State**	**Description**
ServerID	No		Unique identifier for server components. This server identify is unique within a deployment. This ServerID is not known to eStream clients. Its purpose is as a handle to uniquely identify server components.
ServerType	No		Identifies the type of server component. One of the following applies: ▪ Primary Monitor ▪ Backup Monitor

			▪ Application Server ▪ SLiM Server
DbUser	No		User name string required for database connectivity for this server ID
DbPasswd	No		Database password associated with the DbUser
Dsn	No		Data Source Name used to access the database.
PortNum	No		PortNumber used for light-weight messaging listener
MachineID	No		Machine ID is used to get at important machine information needed for all server components such as: ▪ IP address for the machine server component is hosted on ▪ Domain name for the machine ▪ Machines name
AutoReStart	Yes	Any State	Flag indicating that server component process can be restarted automatically without manual intervention
TimeOut	Yes	Any State	Specifies the timeout period for the listener. If the timeout period is reached. The component assumes that it has lost the connection. All Server components have a listener by which they receive instructions from the primary system monitor. Even the monitor has a listener that communicates with the Server Admin UI.

Command Line Utilities:

The *Command Line Utilities* component provides a consistent way to define and process command line arguments. To use this utility, the using component must define a table of arguments, which defines the valid set of arguments, whether or not they are required, and any default values.

Arguments are specified on the command line as name/value pairs. The utility implements the following command line syntax to support the name/value pairs. The argument syntax is defined as follows:

<name>=<value>

name	Name is an alpha-numeric identifier. The Name can be of arbitrary length as supported by the system however shorter names are recommended. Names are case sensitive
value	Any alpha-numeric value. Punctuation characters may also be used. Values are case sensitive

There can be no spaces between the <name>, "=", and the <value> elements. The existence of one or more spaces or tabs delineates separation between arguments on the command line.

Example: server.exe sid=267 dns=oracle user=michaelb passwd=mypasswd

- If a named argument is specified more than once on the command line, subsequent arguments will cause a diagnostic to be issued and the argument will be ignored.
- This utility allows the user to specify default values for arguments. If a default value is defined then the argument will be processed with its default in the event that the argument is not specified on the command line.
- This utility allows the user to tag specific arguments as required. If the required argument is not specified on the command line this utility will raise a diagnostic for the required argument. Not specifying a required argument will cause a fatal error.

The following options are supported:

sid	Server Component Identifier. Each server component within a deployment is uniquely identified via the sid. The sid is a handle into the database for accessing information unique to a specific server component.
dsn	Data Source Name. A data source name is necessary to establish an ODBC connection. Data Source Names are generated by an ODBC administrative tool
dbuser	User name. For database access security, all components need to connect as a specific user.
dbpasswd	password associate with the dbuser

Logging Utilities:

All servers and clients in eStream 1.0 need to log the error and access data. Logging enables component debugging and auditing support.

EStream Framework should provide logging with the following features:

- Each component will have an error and optioanally an access log file. The names of these files would be <component>_error.log and <component>_access.log.
- The files will be located in the <eStream1.0 Root Dir>\logs directory.
- The error log files will have messages with the following priorities:
  - 4-Low : A warning which can be ignored.
  - 3-Medium: A warning which needs to be looked into.
  - 2-High: Recoverable Error in the component.
  - 1-Critical: Fatal Error. Needs admin assistance.
- Logging level should be configurable. The following levels are to be supported.
  - 0: Only errors will be logged. This will be the default level.
  - 1: Errors and Warnings to be logged.
  - 2: Errors, Warnings and Debugging information to be logged.
  - 3: Errors, Warnings and advanced Debugging (like memory dumps, tcp stack dumps etc) to be logged.

- Log Wrapping to be supported. The log files will wrap at a predefined size. On wrapping the following actions will occur:
  - Any existing <logfile>.bak will be deleted from the system.
  - The current <logfile> will be backed to <logfile>.bak
  - The component will continue logging to the <logfile>.

For each eStream client and server component logging the log files (component_error.log and component_access.log) should be written in eStream1.0Root\logs directory. The formats for the log files will be as follows:

Error Log:

[HEADER]
[TimeStamp] [Thread ID] [Priority] [Message]
...
[FOOTER]

An example of this log format would be:

```
*********************************************
Omnishift eStream Application Server
Server Started.
StartTi[redacted] 16:31:19 -0700
IP Address: [redacted]
Logging Level: 3
*********************************************

[14/Aug/2000:16:31:19 -0700] 0 2-High Cannot connect to the database.
Invalid Username/Password.
[14/Aug/2000:16:31:19 -0700] 1 1-Critical Cannot start the HTTP listener
at port 80.
[14/Aug/2000:16:31:19 -0700] 0 1-Critical Shutting down the server.

*********************************************
Omnishift eStream Application Server
Server Stopped.
StopTime:[redacted] 16:35:19 -0700
IP Addres[redacted]
Logging Level: 3
*********************************************
```

Format of Access Log Message:

[HEADER]
[TimeStamp] [Thread ID] [Message]

[FOOTER]

Data type definitions

Server State:

The server components can be in any one of the following states:

State	Description
STOPPED	If a server is in the STOPPED state then the component process is not running.
IDLE	Server component is up and running. The server has been initialized with the common configuration and the messaging system has been enabled. The listener is actively waiting on the System Monitor for transition requests. The server component is not processing any work specific to this servers specialization.
PROCESSING	Server component is actively taking requests and processing work specific to its specialization. ie. serving access tokens, and application file requests.
ERROR	An error has occurred in the system. Unless the server component is configured with **AutoReStart** and ERROR state must be manually cleared by the server-side administrator.

Server State Transitions:

Changes in server component state are initiated either by the *System Monitor* or directly by the server-side administrator for the system monitor. The exception to this is when an error condition is raised by a server component. In this case, the component will initiate the state change itself. The following state transitions are supported:

Action	Description
START_SERVER	Server is expected to be in the STOPPED state. If a server component is configured to support AutoReStart then the ERROR state is also a valid state from which to initiate this action.
STOP_SERVER	Causes the server to exit its process. The server can be stopped from any state.
START_PROCESSING	Causes the server to change from the IDLE state to the PROCESSING state.
STOP_PROCESSING	Causes the server to change from processing to IDLE state.
UPDATE_CONFIG	Request that the server read its configuration from the configuration manager and change its configuration.

RAISE_ERROR	Request that the server go to ERROR state. This causes an error handler to be called. If the error is fatal it will cause immediate termination of the server process.

Finite State Table:

```
FSMTableEntry ServerStateMgr::FSMTable[] =
{
        { START,{{START_SERVER,STOPPED, START_SERVER,NULL},
                {START_PROCESSING, STOPPED ,START_PROCESSING, NULL},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {STOPPED, {{START_SERVER, IDLE, NULL_REQUEST, &StartServer},
                {START_PROCESSING, IDLE, START_PROCESSING, &StartServer},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {IDLE,{{START_PROCESSING, PROCESSING, NULL_REQUEST,
&StartProcessing},
                {STOP_SERVER, STOPPED, NULL_REQUEST, &StopServer},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {UPDATE_CONFIG, IDLE, NULL_REQUEST, &UpdateConfig},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {PROCESSING,{{STOP_PROCESSING, IDLE, NULL_REQUEST,
&StopProcessing},
                {UPDATE_CONFIG,PROCESSING, NULL_REQUEST,
&UpdateConfig},
                {STOP_SERVER, IDLE, STOP_SERVER,  &StopProcessing},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        { ERROR,{{STOP_SERVER, STOPPED,NULL_REQUEST, NULL},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} }

        {NULL_STATE,{{NULL_REQUEST, NULL_STATE, NULL_REQUEST,
        NULL} } }
};
```

Messaging Service Protocol:

A light-weight messaging protocol is needed to facilitate communication between server components. The primary purpose of the messaging protocol is to communicate transition requests to the server components. In response, server components communicate state, status, and load information back to the *System Monitor*.

The messaging protocol supports two primary message types. 1) Requests for the *System Monitor* to perform on other servers. 2) Requests to the server components themselves. These message types are distinguished through the protocol as described below. If the receiver ID and the target ID are identical then the request is for the receiver. If the target is different than the receiver, the message is for the *System Monitor* to enact a request on the target component.

All requests are required to be acknowledged. Without an acknowledgement the message is considered un-received.

OpCode	**senderID**	**receiverID**	**targetID**	**Data**

The following table describes the protocol used by the Server State Manager in its communication with the System Monitor.

OpCode	**Description**	**Data**
0x01	Request for current state	None
0x02	Acknowledgment	▪ Current state ▪ Load info ▪ Status info
0x03	Stop Server request. Acknowledged with 0x02 message	None
0x04	Start Server request. Only valid for System Monitor. Acknowledged with 0x02	None
0x05	Start Processing Request. Acknowledged with 0x02	None
0x06	Stop Processing Request. Acknowledged with 0x02	None
0x07	Update Configuration Request. This is a request for a server component to request its specialized configuration information from the System Monitor and update itself. Acknowledged with 0x02.	None

The messaging protocol used by the configuration manager is described below:

OpCode	**Description**	**Data**
0x01	Request a complete reload of Configuration	
0x02	Request a reload of specific configuration items.	▪ Number of items being requested. ▪ array of names of configuration items.
0x03	Acknowledgement of configuration reload request.	▪ Number of tuples being returned ▪ Flat representation of configuration tu-

		ples

Interface definitions

Server State Manager:

```
class ServerStateMgr
{
private:
        ServerState CurrentState;
        static FSMTableEntry FSMTable[];

public:
        ServerStateMgr(void);
        ~ServerStateMgr(void);

        ServerState SetState(ServerState);
        ServerState GetState(void);
        ServerState ProcessRequest(ServerRequest);
};
```

SetState	<u>**Description:**</u> Sets the current state of the server component. 1. Log the state change request 2. Update the state field within the server component in memory data structures. 3. Send message to requester informing them of the successful state change. Note: **SetState** does not update the database directly as in the orginal design. The database is updated by the *System Monitor* once it has received an acknowledgement. A state transition is not complete until **SetState** returns successfully and the Monitor has update the database. <u>**Input:**</u> state value to set current state to. <u>**Output:**</u> current state after the new value has been set. If an error occurs will go to error state. <u>**Errors:**</u> 1. Invalid state argument 2. Failure to either connect or commit state change to the database.

GetState	<u>**Description:**</u> returns the current state. This function does not read from the database to get the current state. The assumption is that if the server component is up and running and that it maintains a valid state. <u>**Input:**</u> none. <u>**Output:**</u> returns the current state. <u>**Errors:**</u> None. Will always return a valid state.

ProcessRequest	<u>**Description:**</u> request to the Server State Manager to change server state. This routine implements the guts of the state machine.

	1. Get the current state, and transition request 2. Index into the FSM table and continue to transition from state to state until the transition request is satisfied. 3. Each state transition calls the specialized transition routines for each component. 4. Call to **SetState** to complete each state transition. 5. In the case of an error the state machine will process a RAISE_ERROR request which will call the specialized **HandleError** and transition to the ERROR state. **Input:** server transition request. Refer to table of valid requests defined above. **Output:** current state after the request has been completed. **Errors:**

Server Component Framework:

```
class ServerComponent: ServerStateMgr{     // abstract base class
private:
        ErrorInfo*      Error; // maintains error if error was detected
        ServerConfig*   Config;  // holds common configuration
        Connection*     Listener; // messaging         utility
public:
        virtual int StartServer(void); // may be specialized by a server component
        virtual int StopServer(void); // may be specialized
        virtual int StartProcessing(void) = 0; // must be specialized
        virtual int StopProcessing(void) = 0; // must be specialized
        virtual int UpdateConfig(void) = 0; // must be specialized
        virtual int HandleError(void) = 0; // must be specialized
        void Run(Request);
}
```

StartServer	**Description:** Called by the *Server State Manager* when a server component is to be started. The **StartServer** routine is provided as part of the SeverComponent class. It performs the following: 1. Send request to System Monitor to request an update of common configuration information. 2. Apply the configuration information to the server component. 3. Construct a listener connection object and start the message service. 4. Return success or failure. Note: ▪ This routine must return immediately to the main thread. Otherwise the *Server State Manager* will be blocked. ▪ Successful return from the **StartServer** routine will put the server into the IDLE state. **Input:** None. **Output:** Value of 0 if successful else error condition **Errors:** May return negative error condition

StopServer	**Description:** Called by the *Server State Manager.* 1. Perform any necessary cleanup. 2. Send last acknowledgment confirming shutdown to requester 3. Shut down the messaging system and the listener. 4. exit process Note: The monitor will update the database and perform logging. **Input:** None. **Output:** Value of 0 if successful else error condition **Errors:** May return negative error value

StartProcessing	**Description:** Called by the *Server State Manager*. This routine must be defined by each specialized server component. This routine is used to provide all functionality unique to different types of servers.
	1. Spawn a primary processing thread (also known as the boss thread).
	    a. Read server specific configurations unique to this type of server component from the System Monitor
	    b. Spawn worker threads. Depending on the server type this routine does the heavy lifting to either process access tokens and renewals in the case of SLiM server, or process file requests for application servers, or manage and monitor the server components in the case of the *System Monitor*.
	Note:
	▪ This routine must return immediately so that the *Server State Manager* can continue to operate in the main thread.
	▪ This routine may make use of the *Server Configuration Manager* for obtaining specialized configuration information
	Input: None
	Output: Value of 0 if successful else error condition.
	Errors: TBD

StopProcessing	**Description:** Called by the *Server State Manager*. This routine must be defined by the specialized server component type. 1. Reverse all actions performed by the **StartProcessing** routine. All worker threads should be joined or pooled in waiting state. Successful return from this routine will put the server component into the IDLE state. **Input:** None. **Output:** Value of 0 if successful else error condition. **Errors:** TBD

UpdateConfig	**Description:** Called by the *Server State Manager*. This routine must be defined by the specific server component type. The purpose of this routine is apply dynamic configurations or update specialized configurations that are unique to this server component. <may require adding a new state to separate dynamic and static configurations> **Input:** None. **Output:** Value of 0 if successful else error condition. **Errors:** TBD

HandleError	**Description:** Component defined error handling routine to handle errors such as timeouts, etc. This routine will need to handle a number of error cases as are possible by the specialized component. The error information is maintained with

	the ServerComponent class. **Input:** None. **Output:** Integer value designating a handled error or failure. If the error cannot be handled then it is fatal. **Errors:** TBD

Run	**Description:** This routine implements the main processing loop for the server component and runs in the main thread. This routine drives the server component by initiating state requests from the *System Monitor.*
	Note: The *Server State Manager* always runs in the main thread.
	1. Call **ProcessRequest** to transition the server component into the initially requested state.
	2. Enter main processing loop
	    a. Check for requests from the message service.
	    b. Call **ProcessRequest** to service the request.
	    c. Send acknowledgement for the request to the message service. Acknowledgement includes new state, load info, and status.
	Input: Initial Transition Request
	Output: None. This routine should never return
	Errors: None.

Server Component Main Loop:

The following main loop is common to all server components:

```
void ServerComponent::Run(ServerRequest Request)
{
        ProcessRequest(Request);
        while (1)
        {
                Request = Listener->GetRequest();
                ProcessRequest(Request);
                Listener->AckRequest(Request, GetState, GetLoad, GetStatus);

        }
}
```

```
#include "ServerArgs.h"
#include "Server.h"

int main(int argc, char* argv[]) {
        Args = new ArgList();
        Args->ProcessArgList(argv, argc);
        Server = new ServerComponent(GetValue(SID),
```

```
                                GetValue(DNS),
                                GetValue(DBUSER),
                                GetValue(PASSWD));
        Server->Run(START_PROCESSING);
}
```

Command Line Utilities:

```
class NameValuePair
{
        private:
                char* Name;
                char* Value;
        public:
                NameValuePair();
                ~NameValuePair();
                char* GetValue(void);
                char* GetName(void);
                char* SetName(char*);
                char*SetValue(char*);
};
```

```
typedef int (*pFunc)(NameValuePair*);

struct ArgTblEntry
{
        char*   Name;
        bool    Required;
        char*   DefaultValue;
        pFunc   ProcessFunction;
};
```

```
ArgTblEntry const ServerArgsTbl[] = {
        {"sid",             true,   0,          &ProcessSId},
        {"dsn",             true,   0,          &ProcessDsn},
        {"dbuser",          true,   0,          &ProcessDbUser},
        {"dbpasswd",        true,   0,          &ProcessDbPasswd},
        {0,                 0,      0,          0}
};
```

```
typedef vector<NameValuePair*>    ArgVector;

class ArgList
{
        private:
```

```
        ArgVector                  ArgVec;
        const ArgTblEntry*         ArgTbl;

    private:
        NameValuePair*             ParseArg(char* Arg);
        char*                      ParseName(char* Arg);
        char*                      ParseValue(char* Arg);
        int                        ProcessArg(NameValuePair*);
        int                        FinalizeArgs(void);

    public:
                                   ArgList(const ArgTblEntry*);
        int                        ProcessArgList(char* argv[], int argc);
};
```

ProcessArgList	<u>**Description:**</u> Process the entire argument list. In a loop for each argument argv[] … 1. Call **ParseArg** passing in argv[]. 2. **ParseArg** passes the result to **ProcessArg** 3. After processing the entire argument list and exiting the loop call **FinalizeArgs** <u>**Input:**</u> argv and argc as passed into main() entry point <u>**Output:**</u> integer value designating success or failure <u>**Error:**</u>
ParseArg	<u>**Description:**</u> Takes a char* argument and verifies that it follows that name/value syntax defined as <name>=<value> <u>**Input:**</u> Next char* argument on the list <u>**Output:**</u> **NameValuePair**. NULL will be returned in the event of a syntax error <u>**Error:**</u>
ProcessArg	<u>**Description:**</u> This routine performs the semantic analysis of an argument. 1. Look up name in the **ArgTbl** 2. Verify that the value is valid 3. Add the name value pair to a list of processed arguments called **ArgVec** list. 4. If this name value pair already exists in the list then issue a diagnostic. 5. Call the supplied processing function for this argument as specified in the **ArgTbl** <u>**Input:**</u> **NameValuePair** <u>**Output:**</u> Interger value designating success or failure (0 for success, positive integer for other errors) <u>**Error:**</u>
ParseName	<u>**Description:**</u> Verify that the Name part of the argument conforms to being alpha-numeric <u>**Input:**</u> char* Name part of argument <u>**Output:**</u> char* Name else NULL <u>**Error:**</u> None
ParseValue	<u>**Description:**</u> Verify that the Value part of the argument conforms to being alpha-numeric and/or punctuation characters <u>**Input:**</u> char* Value part of argument <u>**Output:**</u> char* Value else NULL <u>**Error:**</u> None
FinalizeArgs	<u>**Description:**</u> Post process the argument list. The purpose of this routine is to validate that all required arguments have been defined on the command line. Also processes and adds default arguments to the **ArgVec**. <u>**Input:**</u> None <u>**Output:**</u> Success or Failure <u>**Error:**</u>

Configuration Manager:

```
class  Tuple {
        string name;
        Value value;
};

class Value {
        int type;
};

class StringValue: public Value{
        string value;
};

class TupleValue: public Value {
        vector <tuple> tupleArray;
};

typedef vector < tuple > ConfigArray;

class ServerConfig {
private:
        ConfigArray Array;
public:
        ServerConfig(serverId, dns, dbuser, dbpasswd); // Initialize from db
        ServerConfig(serverId, string filename); // To initialize from a file.

        ConfigArray* GetConfigArray(int serverId);
        Tuple* FindConfig(string Name);
        int Reload(void);
        Tuple* GetConfig(int serverId ,string Name);
};
```

ServerConfig	**Description:** Constructor for Configuration Manager. 1. Initializes configuration manager. 2. Opens the database and gets configuration array **Input:** Server Id, Data Source Name, Database User name, and database users password. **Output:** None **Errors:**
ServerConfig	**Description:** Constructor for Configuration Manager. 1. Initializes Configuration Manager. 2. Opens configuration file and reads configuration array. **Input:** filename of flat-file configuration. **Output:** None **Errors:**

GetConfigArray	**Description:** Returns the entire configuration for a given server id. This routine always retrieves its information either from the flat file or the database. **Input:** ServerId specifying which server to retrieve configuration for **Output:** Returns a vector holding the configuration or NULL **Errors:**

GetConfig	**Description:** Returns the configuration for the specified name. This routine always retrieves its information either from the flat file or the database. **Input:** ServerId specifying the server to retrieve configuration for and Name of configuration item. **Output:** Configuration Tuple. A Tuple may be a nested Tuple. NULL if an error is encountered. **Errors:**

FindConfig	**Description:** Returns the Tuple specified by the name. This routine does not go to the database or flat-file to get its value. Rather it finds the value in the ConfigArray maintained by the Configuration Manager. **Input:** Name of the configuration item. **Output:** Configuration Tuple. NULL if an error is encountered or the Tuple does not exist in the current configuration. **Errors:**

Reload	**Description:** Reloads the entire configuration from the database or flat-file. This routine may reload its configuration indirectly through the use of the System Monitor. In this case it will make a message request to the monitor and listen for the configuration results. **Input:** None **Output:** integer specifying success or failure. Zero will be returned in the case of Success. A negative value in case of error. **Errors:**

Logging Utilities:

```
class LogManager
{
private:
        char* FileName;
        int MaxFileSize;
        char* ResourceFile; // message catalog file

        char* GetMessage(MsgNum, MsgStr)
public:
        LogManager(ServerId,Size=10);
        LogMessage(MsgStr);
        LogMessage(ThreadId, MsgNum, MsgStr, ...);
```

```
};
```

LogMessage	**Description:** Write message out to log file. There are two forms of LogMessage. The first will write out a message buffer as is (unformatted) bypassing the resource file. The second form will format the message. Both forms of LogMessage always pre-append a time stamp. 1. Lookup message number in the resource file and get message string 2. format the log message using time stamp, thread id, etc. 3. write out message into the log file. **Input:** Thread Id, Message Number, Message String, and variable number of arguments. **Output:** None. **Error:**

GetMessage	**Description:** Routine returns a message string from the resource file for the message number specified. **Input:** Message number, C Locale text string. **Output:** Message string. Either way, Get Message will always pass a return a valid message string by either returning the string from the resource file or by passing back the MsgStr passed in. **Error:** If an error occurs trying to get a message from the resource file, a message will be logged to the error log.

```
class ErrorLog: protected LogManager
{
private:
        LogLevel ErrorLogLevel;
public:
        ErrorLog(ServerId, LogLevel=0, Size=10);
        LogError(ThreadId, ErrorNum, ErrorMsgStr, ...);
};
```

LogError	**Description:** Writes output to error log file. 1. Check that the message level against the current ErrorLogLevel. 2. Format the message and call the long form of **LogMessage** to write the buffer out to the file. **Input:** 1. ThreadId: Thread identifier to help with the debugging process. 2. ErrorNum: Error number used to uniquely identify an error message in the resource file. 3. ErrorMsgStr: Message string which includes stdio like string formatting. 4. …: variable list of arguments to be inserted into the message string per the format. **Output:** None. **Error:**

Testing design

Each of the components that make up the Server Component Framework will be able to be tested independently of the other components. Each component will have a main entry point defined within a testing .exe to accomplish the Unit testing phase.

Testing of the component framework will be done in phases. Each of the phases is described below along with its dependencies.

Phase 1: Unit testing Test basic components that make up the framework. Each components functionality, restrictions, and boundary conditions will be tested. Will allow testing common configurations for a single server component. This round of unit testing will test the integrated component utilities and framework.	**Dependencies:** 1. ServerComponent class 2. ServerStateMgr class 3. ArgList class 4. Logging Utilities 5. Configuration Manager (flat-file)
Phase 2: Unit testing (full functionality) Test full functionality including messaging interfaces and database connectivity.	**Dependencies:** 1. Phase 1 2. Database connectivity 3. Messaging Service
Phase 3: Integration Testing	**Dependencies:** 1. Phase 2 2. Syst[illegible] (including backup) 3. SLiM Server, App Server, Web-Server
Phase 4: Stress Testing See section on stress testing for details	**Dependencies:** 1. Phase 3